Smart Card Authentication and Authorization Framework

BY SAURABH BHATLA

A MASTER'S THESIS IN COMPUTER SCIENCE

Submitted to the Graduate Faculty of Texas Tech University

in partial fulfillment of the requirement for the degree of Master in Science

Approved

_____

Chairperson of the committee

_____

_____

Accepted

_____

Dean of the Graduate School

May 2005

ACKNOWLEDGEMENT

I would like to express my gratitude to all those who helped me to complete this thesis. I am deeply indebted to my supervisor Prof. Dr. Michael W Sobolewski whose help, stimulating suggestions and encouragement helped me during the time of research for this thesis. I would like to give my sincere thanks to Dr. Hector Hernandez and Dr Yu Zhuang for serving on my Thesis committee, for detailed reviews and for their invaluable guidance.

I would also like to thank all my lab mates at the SORCER laboratory for providing a wonderful ambiance which made me feel at home and thus increased my productivity. I am also thankful to the Computer Science department for providing hardware and infrastructure for completing my research.

I dedicate this thesis to my parents and brother for their encouragement, love, support and enthusiasm they have shown all my life. It is because of them that I am able to see this high point of my life.

TABLE OF CONTENTS

ABSTRACT

The service-oriented approach to computing has gained the widespread attention of researchers and the industry. Major initiatives include service-oriented programming (SOP) for constructing software components and service-oriented architectures (SOA) for distributed applications. Software programs developed using SOP can be thought of as mega programs, where the component programs can exchange messages through clearly defined interfaces. SOP environment relies on the exchange of information between various services over various networks.

Services may exchange sensitive information that should only be available for a limited number of persons. Therefore it is necessary that various principals (people, computers, servers) can authenticate themselves. Authentication means that a principal can prove his identity. This can be done by means of secrets, usually cryptographic keys. The process of deciding if user X is allowed to have access to service Y is called authorization. SOP environments may require authorization based on user interaction before he/she is allowed access to the services. Further, if sensible information is sent over an open network, an eavesdropper should not be able to understand the information that is sent and he should not be able to change this information without the receiver detecting this.

Smart cards and the online authentication technology known as Public Key Infrastructure (PKI) seems the perfect solution to achieve this. They are designed to allow

individuals anywhere in the world to identify each other, exchange data in encrypted form and to digitally "sign" documents in ways that cannot later be repudiated.

My research is based on designing a Smart Card based framework for SORCER that will provide user authentication and authorization. This standard security mechanism will not only enforce more consistent security policies, but application developers will be freed from the low-level drudgery of building explicit security controls into their software.

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Walk up to an ATM anywhere in the world, insert your bank card, punch in your PIN, and within minutes you can withdraw local currency from your own account, no matter where you normally bank. Aside from a possible service charge, the transaction is seamless. It's the same as if you were at a branch in your hometown. That's a federated system in action. Out of mutual self-interest, using simple authentication at the point of transaction, participating banks have agreed to trust one another to supply funds from their respective vaults. The banks remain separate entities, but the flow of transactions is shared, creating a federated network. Today, major IT vendors are looking to this same model as a way to enable the next generation of integrated network services. As open federated identity standards mature, IT will be able to deploy sophisticated access controls and to secure multiparty transactions across all types of organizations.

SOA is a conceptual architecture for implementing dynamic e-business. It is a way of designing a software system to provide services to end-user applications or other services through published and discoverable interfaces. Services provide a better way to expose discrete business functions and, therefore, an excellent way to develop applications that support business processes. Identity-based security controls are the natural choice for SOA because they are not dependent on any single application design or technology. Any number of tools could be used to authenticate a user to a given identity, for example, ranging from simple passwords, to digital certificates, to Kerberos,

to biometrics. Individual services need not know anything about the underlying authentication system so long as they are satisfied with the validity of the user's digital identity. PKI is perfect for service-to-service transactions on the Internet. An individual can use a PKI-enabled digital certificate to encrypt and sign messages, and the person on the other end, anywhere on the Internet, can be certain who sent the message and that the contents were not altered.

The issue involves questions as to where user identity should actually reside, the role of technology versus the role of trust, and how open standards can ever hope to rationalize the matrix of permissions required to share user information across an endless diversity of systems and organizations. We have to make sure that the private key used to guarantee the identity of the certificate-holder is not stolen by a hacker accessing a certificate stored on a computer. Security provided by digital certificates is only as good as the security provided for the storage and use of the private keys. Smart cards are small, tamper-resistant devices providing users with convenient storage and processing capability. The smart cards are suitable for cryptographic implementations because they contain many security features that enable the protection of sensitive cryptographic data and provide for a secure processing environment. The protection of the private key is critical for digital signatures. This key must never be revealed. The smart card protects the private key, and many consider the smart card an ideal cryptographic token.

1.1 Challenges

We want the service and the smartcard holder to communicate in such a way that the following security requirements are met:

- *Authenticity* means that the server and the smartcard holder trust each others identity.
- *Authorization* guarantees that the authenticated person has the right to access the service or data.
- *Integrity* requires that changing a message by an intruder will be detected by the receiver.
- *Confidentiality* assures protection of a message in such a way that unauthorized principals are not able to interpret the information in the message.
- *Non-repudiation* guarantees that the sender of the message cannot deny that he/she sent it at a later point in time.

When a certificate is presented to an entity as a means of identifying the certificate holder (the principal of the certificate), it is useful only if the entity receiving the certificate trusts the issuer, which is often referred to as the certification authority (CA). When we trust a certification authority, that means we have confidence that the certification authority has the proper policies in place when evaluating certificate requests and will deny certificates to any entity that does not meet those policies. In addition, we trust that the certification authority will revoke certificates that should no longer be considered valid by publishing an up-to-date certificate revocation list Certificate revocation lists are considered valid until they expire. The idea is that if the user trusts the

CA and can verify the CA's signature, then he can also verify that a certain public key does indeed belong to whoever is identified in the certificate.

1.2 Motivation

The problem is that when it comes to security, developers have historically been forced to repeatedly reinvent the wheel. Whereas modern programming languages such as C#, Java, and Python incorporate levels of abstraction that free developers from thinking about low-level tasks such as memory management, there are no such standard facilities for the basic functions of user authentication and authorization. By building standard security mechanisms we can not only enforce more consistent security policies, but application developers are freed from the low-level drudgery of building explicit security controls into their software.

In my thesis I intend to use the features of Public-key cryptography and Jini architecture to design a Smart Card Authentication and Authorization Framework (SCAF) that would integrate into Service Oriented Computing Environment (SORCER) and hence enable secure service oriented computing. With SCAF it would be possible for the developer to leverage the framework and provide security in SOA. The following sections provide a brief background of Public-key cryptography and various technologies used to perform service oriented computing.

1.3 Thesis Organisation

In traditional remote procedure call systems like CORBA or DCOM the connecting fiber between two programs are the client-side stub and the server-side skeleton [1]. Stub starts communication to the server by opening up a communication channel, it then converts all the arguments to a form that can be transmitted across the channel, and sends those converted arguments. When response comes from the server, the stub converts any return values from their wire representation to the internal form used in the process, and returns those results back to the process that originated the call. On server side, similar functionality is provided skeleton code. It receives the information transmitted by a stub, converts the received information into a form that the server program understands. It then calls the appropriate server program and translates the results send by server program into a form that can be transmitted over the wire, and sends them back to the calling client.

For generating code for stub and skeleton, a definition of the interface between the client and the service written in some neutral declarative language is given to a compiler. The source code hence generated it compiled by native compilers on the machine it is going to be executed.

Systems based on protocols also have their limitations. The information that can be represented in the protocol is limited to the kinds of data found in languages that the protocol is translated into. To satisfy the needs of client and server, all requirements have to be fit into a single protocol in protocol based systems. The biggest problem with such systems is their rigidity once deployed. If there is a change at any of the ends the change

needs to be propagated the other end. In current environment, service gets redefined all the time, which makes requirement for continuous updates a serious problem for protocol-based systems.

These limitations resulted in environment in which code can be dynamically loaded into a running process no matter what the underlying processor or operating system. In Java technology environment this functionality is provided by RMI. The code that is used by the client to access a service is not deployed on the client but is downloaded dynamically. Stubs, in RMI are references that implement all the interfaces implemented by the service that different clients may want to use. JINI based on RMI semantics, takes it a notch further by providing stubs as proxies which can be downloaded from a Lookup Service. This means that the client does not even need to know the location of the service and proxy to service can be found dynamically. JINI network technology is explained in chapter 2.

This dynamic discovery, availability of services and dynamic downloading of code on the requestor side has resulted in the inception of a new architecture for distributed systems which is based on peer-to-peer communication instead of client-server model. Such architecture is known as Service Oriented Architecture which can be simply generalized as a collection of services communicating with each other. SOA basically comprises of simple and ubiquitous set of interfaces universally available for all service providers and consumers. Service ORiented Computing EnviRonment (SORCER) is one such federated grid infrastructure that aims to provide a service

oriented environment which can handle the needs of any distributed scalable network centric system. SORCER framework and design is explained in chapter 3.

Java(TM) security technology provides a safe environment to run potentially untrusted code downloaded from the public network. Fine-grained access controls can be placed upon critical resources with regard to the identity of the running applets and applications, which are distinguished by where the code came from and who signed it. As Java is being widely used in a multi-user environment it is required to enforce access controls based on the identity of the user who runs the code. The Java Authentication and Authorization Service (JAAS) is designed to provide a framework and standard programming interface for authenticating users and for assigning privileges. Chapter 4 contains the detailed explanation of entities in JAAS and how authentication and authorization is achieved.

SOAs are the conceptual architectures for implementing dynamic e-business. Identity-based security controls are the natural choice for SOA because they are not dependent on any single application design or technology. Any number of tools could be used to authenticate a user, for example, ranging from simple passwords, to digital certificates, to Kerberos, to biometrics. With the need for information security in today's digital systems growing, cryptography has become one of its critical components. Digital signatures are one of the many uses of cryptography. Public-key cryptography allows one to digitally sign and encrypt information transacted between parties. Public Key Infrastructure (PKI) uses this technology and adds authentication and non-repudiation of the information regarding the parties concerned. Public Key Cryptography Standards

(PKCS) is a suite of protocols and algorithms and some of the standards used in this thesis are discussed in chapter 5.

The problem is that more and more applications can be secured with such crude things like certificates and keys but it is really difficult to setup PKIs and it is really expensive too because flexible trust center software for Unix is expensive. The goal of OpenCA is the production of an open source trust center system to support the community with a good, inexpensive and future-proof solution for their base infrastructure. Chapter 6 provided an in depth detail of the components and hierarchies of OpenCA.

The most important issue in providing security is to where user identity should actually reside. User credentials saved on servers or in files are not completely safe and are not portable. They are not intrusion protected and are not portable. It is required to share user information across an endless diversity of systems and organizations. Security provided by digital certificates is only as good as the security provided for the storage and use of the private keys. Smart cards (discussed in chapter 7) are small, tamper-resistant devices providing users with convenient storage and processing capability and they contain many security features that enable the protection of sensitive cryptographic data and provide for a secure processing environment. In order to use a smart card, we need to be able to read the card and communicate with it using an application. OpenCard provides a framework for this by defining interfaces that must be implemented. The OpenCard framework defines several of these interfaces. Once these interfaces are

implemented, we can use other services in the upper layers of the API, overview of which is given in chapter 7.

Smartcards typically vary from release to release so the middleware, that communicates with the card and exports the card's functionality to the host, generally is in constant change. Each card must have its own CSP (crypto/card service provider) on the host which has large support problems. Chapter 8 gives an overview of MUSCLE applet approach, using which only one host CSP be written for the middleware, thus reducing the time spent migrating to new card releases and vastly reducing the number of CSP's on the host. The MUSCLE applet has to be loaded on the card with a static application identifier (AID) and the host based CSP will communicate to the card through this applet.

As explained earlier, this research is based on the development of Smart Card Authentication and Authorization Framework which is a Smart Card based framework for SORCER that provides user authentication and authorization. This standard security mechanism enforces more consistent security policies and application developers are freed from the low-level drudgery of building explicit security controls into their software. Chapter 9 elaborates on the Objective/Approach, benefits, design, implementation and validation of SCAF.

CHAPTER 2

JINI NETWORK TECHNOLOGY

2.1 Technology

2.1.1 Introduction

Jini is a set of Java classes and specifications that are platform independent and is built on top of java Technology as shown in the Figure 2.1. It aids the construction of distributed systems where scale, rate of change and complexity of interactions within and between networks are extremely important and cannot be satisfactorily addressed by existing technologies [2]. The Jini architecture specifies a way for clients and services to find each other on the network and to work together to get a task accomplished. Jini technology provides a flexible infrastructure for interactions between clients and services regardless of their hardware or software implementations.

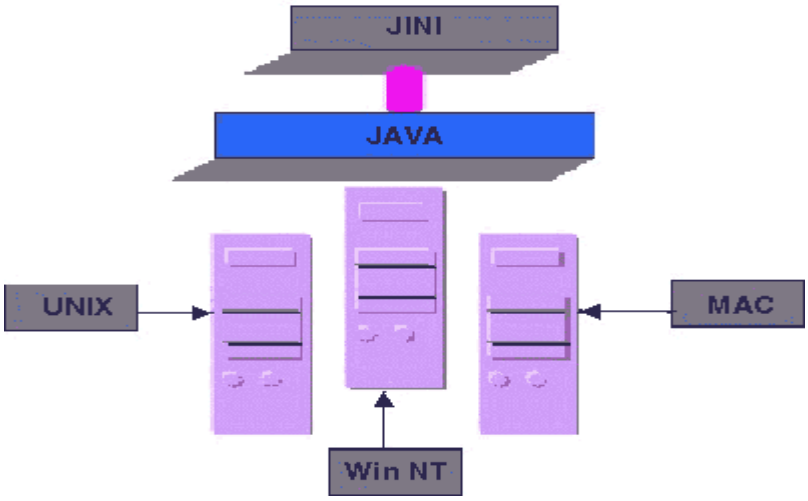


Figure 2.1. Jini-Platform Independent

Jini has several properties which are inherited from Java to support service-oriented architecture. They are:

- Homogeneity
- A single type system
- Serialization
- Code Downloading
- Safety and Security

Jini architecture makes the entire network of services adaptable to changes in the network by using objects that move around the network. The Jini architecture specifies a way for clients and services to find each other on the network and to work together to get a task accomplished. Service providers supply clients with portable Java technology-based objects that give the client access to the service. Since the client only sees the Java object provided by the service, network interaction can use any type of networking technology such as RMI, CORBA, or SOAP. For example, a Jini enabled printer can provide print service to the entire network. Jini services can be either hardware based or software based in nature [3]. This makes it a dynamic environment where any service can enter or leave the network at anytime and provides constructs that make administration of services simple.

Jini system is a collection of clients and services communicating by means of Jini protocols. A Jini system consists of a set of components that provides an infrastructure for federating services in a distributed system, a programming model that supports and encourages the production of reliable distributed services and services that can be made

part of a federated Jini system and which offer functionality to any other member of the federation.

2.1.2 Features

Jini was designed with the goal of making services accessible to everybody on network. Jini enabled device can access any service that becomes available on the network in type-safe and robust way. The features of Jini architecture are explained below:

Connect anything, anytime, anywhere: It provides an infrastructure that helps different network users to discover, join, and participate in any network community spontaneously.

Network Plug and work: It makes available a new service to all the users without any configuration and installation hassles.

Abstraction of Hardware/Software distinction: It provides architecture centered on a service network instead of a computer network or device network.

Promote Service Based Architecture: Applications created for stand alone purposes can be made available as services in the network by certain deploying mechanisms. This enables service based architecture wherein all applications can be considered as services for clients in the network.

Simplicity: Jini services provide a simple generic framework. Services are designed in a way that they are reusable and can be modified according to the users needs.

2.1.3 Architecture

The Jini system extends the Java Application Environment from a single virtual machine to a network of machines. Jini Architecture [4] provides an infrastructure for defining, advertising and finding services in a network. As both code and data can move from machine to machine the Java application environment provides a good computing platform for distributed computing. The Jini architecture adds mechanisms that allow fluidity of all components in a distributed system, extending the easy movement of objects to the entire networked system. The environment provides built in security for downloaded code from one machine to another.

Jini technology blurs the distinction between devices and software by concentrating on the services that devices provide. Devices, whether they are pocket-sized, consumer electronic items, desktop computers, or industrial machinery, provide services that can be utilized by clients. These services can be unified by a Jini network. Jini technology brings object-orientation to the network. Clients of a service need only know the interface of that service written in the Java programming language (*Java interface*) to use it. The details of the service that implements the interface are hidden from the client.

- Services

The most important concept within the Jini architecture is that of a *service*. A service is an entity that can be used by a person, a program, or another service. A service may be a computation, storage, a communication channel to another user, a software filter, a hardware device, or another user. Two examples of services are printing a

document and translating from one word-processor format to some other. Members of a Jini system federate in order to share access to services. Services are defined via an interface, and the implementation of a proxy supporting the interface that will be seen by the service client will be uploaded into the lookup service by the service provider. This implementation is then downloaded into the client as part of that client finding the service. This service-specific implementation needs to be code written in the Java programming language to ensure portability.PA Jini system consists of services that can be collected together for the performance of a particular task. Services may make use of other services, and a client of one service may itself be a service with clients of its own. The dynamic nature of a Jini system enables services to be added or withdrawn from a federation at any time according to demand, need, or the changing requirements of the workgroup using it.

Services in a Jini system communicate with each other by using a *service protocol*, which is a set of interfaces written in the Java programming language. The Jini system defines these set of protocols for services to interact with each other.

- Lookup Service

Services are found and resolved by a *lookup service*. The lookup service is the central bootstrapping mechanism for the system and provides the major point of contact between the system and users of the system. A lookup service maps interfaces indicating the functionality provided by a service to sets of objects that implement the service.

A service is added to a lookup service by a pair of protocols called *discovery* and *join*--first the service locates an appropriate lookup service (by using the *discovery* protocol), and then it joins it (by using the *join* protocol).

- Java Remote Method Invocation (RMI)

Java Remote Method Invocation (Java RMI) enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines*, possibly on different hosts. RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism. Fundamentally, RMI is a Java-programming-language-enabled extension to traditional remote procedure call mechanisms. RMI allows not only data to be passed from object to object around the network but full objects, including code. Much of the simplicity of the Jini system is enabled by this ability to move code around the network in a form that is encapsulated as an object.

Communication between services can be accomplished by Remote Method Invocation (RMI). The infrastructure to support communication between services is not itself a service that is discovered and used but is, rather, a part of the Jini technology infrastructure. RMI provides mechanisms to find, activate, and garbage collect object groups.

- Security

The design of the security model for Jini technology is built on the twin notions of a *principal* and an *access control list*. Jini services are accessed on behalf of some entity--

the principal--which generally traces back to a particular user of the system. Services themselves may request access to other services based on the identity of the object that implements the service. The granting of access to a service depends on the contents of an access control list that is associated with the object.

- Leasing

Access to services in the Jini system environment is *lease* based. A lease is a grant of guaranteed access over a time period. Each lease is negotiated between the user of the service and the provider of the service as part of the service protocol: A service which is requested for some period is granted access for some period, presumably taking the request period into account. If a lease is not renewed before it is freed--either because the resource is no longer needed, the client or network fails, or the lease is not permitted to be renewed--then both the user and the provider of the resource may conclude the resource can be freed.

Leases are either exclusive or non-exclusive. Exclusive leases insure that no one else may take a lease on the resource during the period of the lease; non-exclusive leases allow multiple users to share a resource.

- Transactions

A series of operations, either within a single service or spanning multiple services, can be defined as a *transaction*. The Jini Transaction interfaces supply a service protocol needed to coordinate a *two-phase commit*. The implementation of a transaction is left up to the service using the interfaces.

- Events

The Jini architecture supports distributed *events*. An object may allow other objects to register interest in events in the object and receive a notification of the occurrence of such an event. This enables distributed event-based programs to be written with a variety of reliability and scalability guarantees.

2.2 Components

The components of the Jini system can be segmented into three categories: *infrastructure*, *programming model*, and *services*. The infrastructure is the set of components that enables building a federated Jini system, while the services are the entities within the federation. The programming model is a set of interfaces that enables the construction of reliable services, including those that are part of the infrastructure and those that join into the federation.

2.2.1 Infrastructure

The Jini technology infrastructure defines the minimal Jini technology core. The infrastructure includes the following:

- A distributed security system, integrated into RMI, which extends the Java platform's security model to the world of distributed systems
- The discovery/join protocol, a service protocol that allows services (both hardware and software) to discover, become part of, and advertise supplied services to the other members of the federation

- The lookup service, which serves as a repository of services. Entries in the lookup service are objects in the Java programming language; these objects can be downloaded as part of a lookup operation and act as local proxies to the service that placed the code into the lookup service

2.2.2 Programming model

Entries in the lookup service are leased, allowing the lookup service to reflect accurately the set of currently available services. When services join or leave a lookup service, events are signaled, and objects that have registered interest in such events get notifications when new services become available or old services cease to be active. The programming model rests on the ability to move code, which is supported by the base infrastructure.

Both the infrastructure and the services that use that infrastructure are computational entities that exist in the physical environment of the Jini system. Services also constitute a set of interfaces, which define communication protocols that can be used by the services and the infrastructure to communicate between themselves.

These interfaces, taken together, make up the distributed extension of the standard Java programming language model that constitutes the Jini programming model. Among the interfaces that make up the Jini programming model are the following:

- The leasing interface, which defines a way of allocating and freeing resources using a renewable, duration-based model

- The event and notification interface, which is an extension of the event model used by Java Beans components to the distributed environment that enables event-based communication between Jini services
- The transaction interfaces, which enable entities to cooperate in such a way that either all of the changes made to the group occur atomically or none of them occur

2.2.3 Services

Services are objects written in Java programming language. A service has an interface which defines the operations that can be requested of that service. Some of these interfaces are intended to be used by programs, while others are intended to be run by the receiver so that the service can interact with a user. The type of the service determines the interfaces that make up that service and also define the set of methods that can be used to access the service. A single service may be implemented by using other services. Services form the interactive basis for a Jini system, both at the programming and user interface levels

➢ Discovery and Lookup Protocols

The heart of the Jini system is a trio of protocols called *discovery*, *join*, and *lookup*. A pair of these protocols--discovery/join--occurs when a device is plugged in. Discovery occurs when a service is looking for a lookup service with which to register. Join occurs when a service has located a lookup service and wishes to join it. Lookup occurs when a client or user needs to locate and invoke a service described by its

interface type (written in the Java programming language) and possibly, other attributes. The following diagram outlines the discovery process.

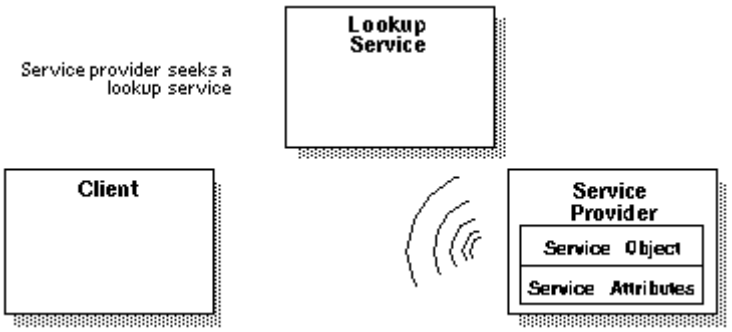


Figure 2.2. Discovery

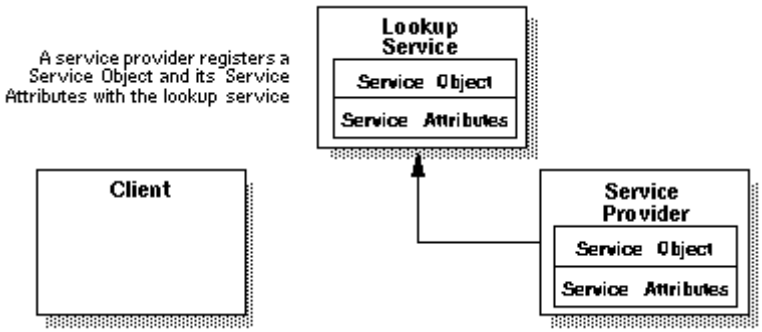


Figure 2.3. Join

Discovery/Join is the process of adding a service to a Jini system. A service provider is the originator of the service--a device or software, for example. First, the service provider locates a lookup service by multicasting a request on the local network for any lookup services to identify themselves (discovery, Figure 2.2). Then, a service object for the service is loaded into the lookup service (join, Figure 2.3). This service object contains the Java programming language interface for the service including the

methods that users and applications will invoke to execute the service, along with other descriptive attributes.

Services must be able to find a lookup service; however, a service may delegate the task of finding a lookup service to a third party. The service is now ready to be looked up and used, as shown in the following diagram (Figure 2.4).

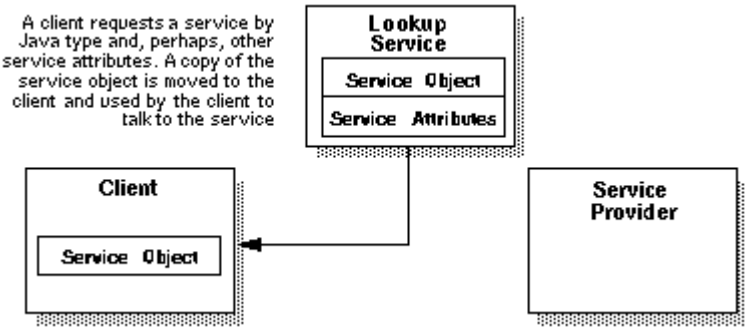


Figure 2.4. Lookup

A client locates an appropriate service by its type--that is, by its interface written in the Java programming language--along with descriptive attributes which are used in a user interface for the lookup service. The service object is loaded into the client.

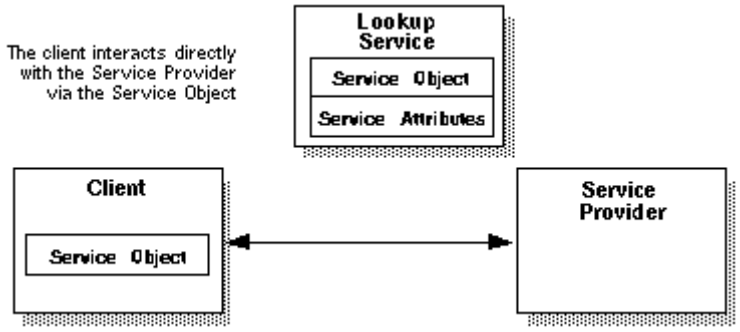


Figure 2.5. Service Invocation

The final stage is to invoke the service, as shown in the following diagram (Figure 2.5). The service object's methods may implement a private protocol between itself and the original service provider. Different implementations of the same service interface can use completely different interaction protocols.

The ability to move objects and code from the service provider to the lookup service and from there to the client of the service gives the service provider great freedom in the communication patterns between the service and its clients. This code movement also ensures that the service object held by the client and the service for which it is a proxy are always synchronized, because the service object is supplied by the service itself. The client only knows that it is dealing with an implementation of an interface written in the Java programming language, so the code that implements the interface can do whatever is needed to provide the service. Because this code came originally from the service itself, the code can take advantage of implementation details of the service known only to the code.

The client interacts with a service via a set of interfaces written in the Java programming language. These interfaces define the set of methods that can be used to interact with the service. Programmatic interfaces are identified by the type system of the Java programming language, and services can be found in a lookup service by asking for those that support a particular interface. Finding a service this way ensures that the program looking for the service will know how to use that service, because that use is defined by the set of methods that are defined by the type.

Programmatic interfaces may be implemented either as RMI references to the remote object that implements the service, as a local computation that provide all of the service locally, or as some combination. Such combinations, called *smart proxies*, implement some of the functions of a service locally and the remainder through remote calls to a centralized implementation of the service. A user interface can also be stored in the lookup service as an attribute of a registered service. A user interface stored in the lookup service by a Jini service is an implementation that allows the service to be directly manipulated by a user of the system. In effect, a user interface for a service is a specialized form of the service interface that enables a program, such as a browser, to step out of the way and let the human user interact directly with a service. In situations where no lookup service can be found, a client could use a technique called *peer lookup* instead. In such situations, the client can send out the same identification packet used by a lookup service to request service providers to register. Service providers will then attempt to register with the client as though it were a lookup service. The client can select those services it needs from the registration requests it receives in response and drop or refuse the rest.

CHAPTER 3

SERVICE ORIENTED COMPUTING

3.1 Service – Oriented program (SO Programming)

Service Oriented Programming (SO programming) is a paradigm for distributed computing built over Object Oriented Programming (OO programming) paradigm emphasizing the point that problems can be modeled in terms of services rather than objects. SO programming differs from OO programming by focusing on what things can do whereas OO programming focuses on what things are and how they are constructed. SO Programming defines set of core principles to maintain interoperability of services over time.

The object-oriented paradigm, defines a system as a collection of interacting active objects. These objects do things and know things, or stated equivalently they have functions and data that complement each other. Usually an object-oriented system creates its own object space instead of accessing a data repository. This object space constitutes an object-oriented program. The execution of the object-oriented program is a collection of dialoguing objects (sending and receiving messages).

Building on the object-oriented paradigm is the service-oriented paradigm, in which the objects are distributed, or more precisely they are network objects and play some predefined roles. A service provider is an object that accepts messages from service requestors to execute an item of work – a task. The task object is a service request – a kind of elementary service executed by a service provider. A service broker is a

specialized service provider that executes a job – a compound request in terms of tasks and other jobs. The job object is a service-oriented program that is dynamically bound to all relevant and currently available service providers on the network. This collection of service providers dynamically identified by a broker is called a job federation. This federation is also called a job space. While this sounds similar to the object-oriented paradigm, it really isn't. In the object-oriented paradigm the object space is a program itself; here the job space is the execution environment for the job itself and the job is a service-oriented program that federates relevant providers at runtime. This changes the game completely. In the former case the object space is a virtual machine, but in the latter case the job space is the virtual federating network. This runtime federation is the jobs' execution environment and the job object is a service-oriented program. In other words, we apply the object-oriented concepts directly to the network in the service-oriented paradigm. Tasks and jobs as elementary and compound service-oriented programs, respectively, are called exertions.

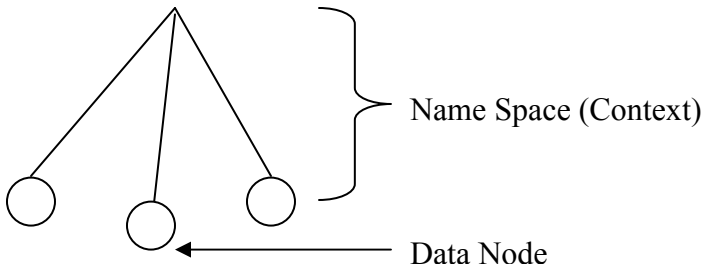


Figure 3.1. Tree structure of context nodes

A *context model* is the exertion's data structure and is based on the percept calculus knowledge representation scheme. It forms the essential structure of the data being processed as specified by the exertion's interface and operation. It is represented as

a tree-like structure of context nodes (Zhao, 199) as shown in figure 3.1 It is represented by the *ServiceContext* interface or alternatively can be represented in XML when used across heterogeneous programming environments. The *context* denotes an application domain namespace, and a context model is its context with data nodes as leaf nodes appended to its context paths. A context path is a name for a data in its leaf node. The leaf node might contain any object and in particular an object that represents a file, for example a URL. A special container object called *ServiceNode* acts as a wrapper that holds a reference to a remote document object available for example from the File Store provider (Sobolewski, 2003).

In the service grid environment two types of basic exertions are defined: *tasks* and *jobs*. A task is the *atomic exertion* that is defined by its *context model* (data), and by its *method* (operation). An exertion method defines a service provider (grid object) to be bound to in runtime. This network object provides the business logic to be applied to the exertion context model as specified in the exertion's method.

3.2 Service Oriented Computing Environment(SORCER)

3.2.1 Framework

The P2P service-oriented framework developed in this work targets complex business and engineering transactions. A transaction is composed of a sequence of activities with specific precedence relationships. The grid contains service providers that offer one or more services to other peers on the overlay network. Service providers do not have mutual associations prior to the transaction; they come together (federate)

dynamically for a specific transaction. Each provider in the federation performs its services in a choreographed sequence. Once the transaction is complete, the federation dissolves and the providers disperse and seek other transactions to join. The architecture is service centric in which a service is defined as an independent self-sustaining entity performing a specific network activity. Each service is defined by a well-known public interface. A service provider that plans to offer a service implements its interface or multiple interfaces (services) to be eligible for participating in federations.

The same provider can provide multiple services in the same federation and different providers can provide the same service in different federations. The service grid is dynamic in which new services can enter the overlay network and existing services can leave the network at any instance. The service-based architecture is resilient, self-healing, and self-managing. The key to the resilience is the transparency of search and seamless substitution of one service with another. The architecture allows services to share data by using specialized data services or a shared data repository (distributed file store). The architecture also allows asynchronous execution of activities such that an activity can wait for a service to be available.

The architecture uses Jini network technology [24] and Java Spaces technology [25] for implementing the service-based overlay network described above. However, the proposed service-oriented architecture is abstract and can be implemented using any distributed network technology that provides support for dynamic discovery of resources and a rich software development environment.

3.2.2 Design

The core of the architecture consists of service providers and service brokers interacting with lookup registries, a catalog of services, and exertion shared space. In general, a service provider executes a task, and a service broker executes a job. While executing a job, the service broker coordinates exertion execution within the nested transaction. It interprets the transaction map supplied by the service requestor and completes the nested exertions. A UML-diagram showing the framework of the system developed is illustrated in Figure 3.2.

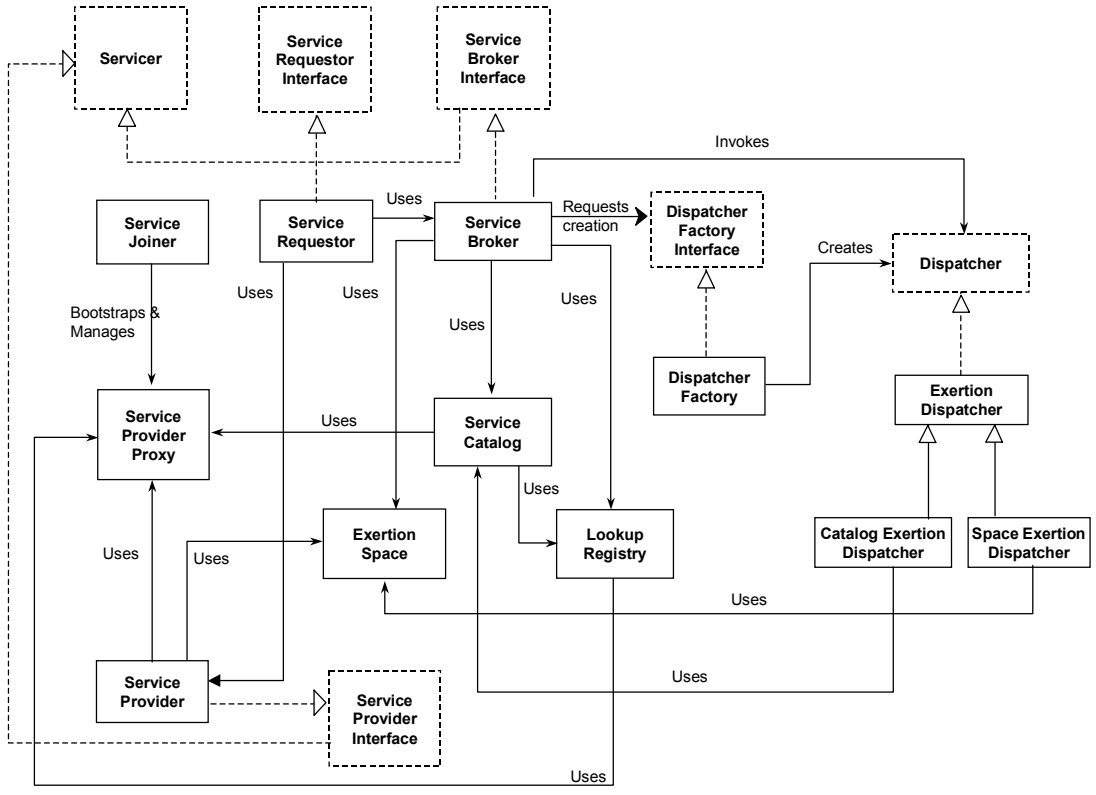


Figure 3.2. UML-Diagram for service-based framework to support nested transactions

At the start of the transaction the service broker reads all the exertions in the transaction and executes those exertions, which have no precedence relationships. At each step it executes the services for which all the precedence relationships have been satisfied (services complete). Whenever it gets a notification of a service being completed it evaluates the remaining unfinished activities and invokes one or more exertions based on their precedence relationships.

The service broker, by using an appropriate exertion dispatcher, can directly access the service provider through a service catalog and select a provider or drop the exertion into Exertion Space for the first available provider to process the request. While the service broker is servicing a job, a nested job within the job being currently serviced can be executed locally, or can be dropped into the Exertion Space, or passed on directly to another service broker. Another available service broker can then federate and collaborate in the job execution by executing the nested job, and so on. Thus not only can the service providers federate to execute a job for a particular service broker, but the service brokers can also federate along with other service providers. The federated brokers with the originating broker execute the nested jobs while the regular service providers execute all the tasks within all jobs including the originating one. A service broker uses a factory design pattern to request a relevant exertion dispatcher that matches the control structure of the executed job.

Two main types of exertion dispatchers are used: a catalog exertion dispatcher and a space exertion dispatcher. The catalog exertion dispatcher finds needed service

providers using the service catalog. The space exertion dispatcher drops exertion into the exertion space to be picked up by matching available service providers. When the exertion is picked up from the space and it is executed by a matching provider then the provider returns it into the space and the space exertion dispatcher gets it back from the space for the service broker. The service grid also defines a common service provider interface (Provider that extends the top level interface Servicer) and a set of utilities to create and register providers with the grid as service peers. A Service Joiner is used for bootstrapping service providers and also for maintaining leases on registered proxies.

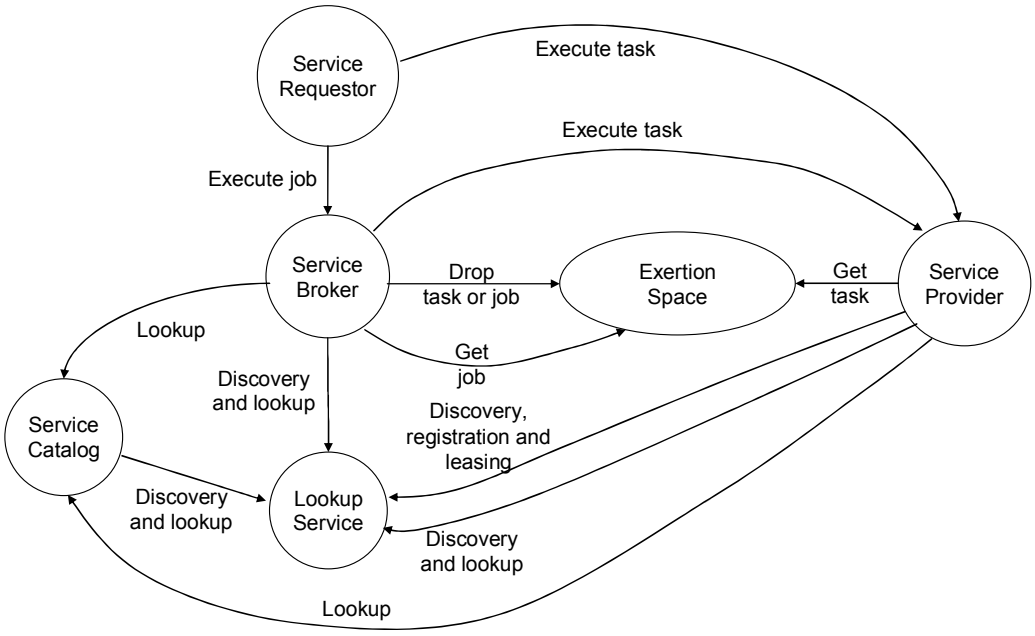


Figure 3.3. Operation of Service Broker and Service Provider

Figure 3.3 shows the different ways in which a provider (the service broker or service provider) can submit requests to the providers. For a direct connection to the service provider the provider can either use discovery to find a lookup service or use a Service Catalog provider for selecting a service. The lookup service caches all the proxies

for services that have registered with it for a particular group(s) of services. The Catalog provider is a service-grid cache that periodically polls all relevant lookup services and maintains a cache of all the proxies that are registered with the lookup services for a particular group or groups of services.

Thus, multiple service catalogs may be used for different logical overlay sub networks. The provider has to discover lookup services each time it needs to use them where as it finds one of required catalogs only once when it (provider) is instantiated and then the Catalog continues service discovery for the provider. In case the provider finds an available service using a lookup registry or the Catalog, a proxy for the service is downloaded on to the provider who invokes the service by calling service (Exertion). Alternately the provider submits the service request to an Exertion Space that holds the request and waits for a matching service provider to accept the exertion. This is essential so that the transaction does not have to abort due to non-availability of a service. This also helps in better load balancing of the services since available providers will act at their own pace to process the exertions in the space. A notification management framework (Lapinski 2002) based on a notification provider allows federated providers notify the service requestor on their collaborative actions. Additionally the File Store provider (Sobolewski 2003) allows federated providers to share exertion input as well as output data is a uniform service-oriented way.

CHAPTER 4

JAVA AUTHENTICATION AND AUTHORIZATION SERVICE (JAAS)

4.1 Introduction

Java(TM) security technology originally focused on creating a safe environment to run potentially untrusted code downloaded from the public network [5]. In Java(TM) Platform, fine-grained access controls can be placed upon critical resources with regard to the identity of the running applets and applications, which are distinguished by where the code came from and who signed it. However, the Java platform still lacked the means to enforce access controls based on the identity of the user who runs the code. Java is being widely used in a multi-user environment. For example, an enterprise application or a public Internet terminal must deal with different users, either concurrently or sequentially, and must grant these users different privileges based on their identities. The Java Authentication and Authorization Service (JAAS) is designed to provide a framework and standard programming interface for authenticating users and for assigning privileges.

The JAAS infrastructure can be divided into two main components: an *authentication* component and an *authorization* component. The JAAS authentication component provides the ability to reliably and securely determine who is currently executing Java code, regardless of whether the code is running as an application, an applet, a bean, or a servlet. The JAAS authorization component supplements the existing Java 2 security framework by providing the means to restrict the executing Java code

from performing sensitive tasks, depending on its code source and depending on who was authenticated.

JAAS authentication is performed in a pluggable fashion. This permits Java applications to remain independent from underlying authentication technologies. Therefore new or updated authentication technologies can be plugged under an application without requiring modifications to the application itself. Applications enable the authentication process by instantiating a LoginContext object, which in turn references a Configuration to determine the authentication technology, or LoginModule, to be used in performing the authentication. Typical LoginModules may prompt for and verify a username and password. Others may read and verify a voice or fingerprint sample.

Once the user executing the code has been authenticated, the JAAS authorization component works in conjunction with the existing Java 2 access control model to protect access to sensitive resources. Unlike in Java 2, where access control decisions are based solely on code location and code signers (a CodeSource), in JAAS access control decisions are based both on the executing code's CodeSource, as well as on the user running the code, or the Subject. JAAS policy merely extends the Java 2 policy with the relevant Subject-based information. Therefore permissions recognized and understood in Java 2 (java.io.FilePermission and java.net.SocketPermission, for example) are also understood and recognized by JAAS. Furthermore, although the JAAS security policy is physically separate from the existing Java 2 security policy, the two policies, together, form one logical policy.

4.2 Subjects and Principals

Users often depend on computing services to assist them in performing work. Furthermore services themselves might subsequently interact with other services. JAAS uses the term; subject, to refer to any user of a computing service [6][7]. Both users and computing services, therefore, represent subjects. To identify the subjects with which it interacts, a computing service typically relies on names. However, subjects might not have the same name for each service and, in fact, may even have a different name for each individual service. The term, principal, represents a name associated with a subject [6]. Since subjects may have multiple names (potentially one for each service with which it interacts), a subject comprises a set of principals.

Principals can become associated with a subject upon successful authentication to a service. Authentication represents the process by which one subject verifies the identity of another, and must be performed in a secure fashion; otherwise a perpetrator may impersonate others to gain access to a system. Authentication typically involves the subject demonstrating some form of evidence to prove its identity. Such evidence may be information only the subject would likely know or have (a password or fingerprint), or it may be information only the subject could produce (signed data using a private key).

4.3 Credentials

Services can also associate other security-related attributes and data with a subject in addition to principals. JAAS refers to such generic security-related attributes as credentials. A credential may contain information used to authenticate the subject to new

services. Such credentials include passwords, Kerberos tickets, and public key certificates (X.509, PGP, etc.), and are used in environments that support single sign-on. Credentials might also contain data that simply enables the subject to perform certain activities. Cryptographic keys, for example, represent credentials that enable the subject to sign or encrypt data.

JAAS divides each subject's credentials into two sets. One set contains the subject's public credentials (public key certificates, Kerberos tickets, etc). The second set stores the subject's private credentials (private keys, encryption keys, passwords, etc). To access a subject's public credentials, no permissions are required. However, access to a subject's private credential set is security checked.

4.4 Authentication

Depending on the security parameters of a particular service, different kinds of proof may be required for authentication. The JAAS authentication framework is based on PAM [8], and therefore supports an architecture that allows system administrators to plug in the appropriate authentication services to meet their security requirements. As new authentication services become available or as current services are updated, system administrators can easily plug them in without having to modify existing applications.

The JAAS LoginContext class represents a Java implementation of the PAM framework. The LoginContext consults a configuration that determines the authentication service, or LoginModule, that gets plugged in under that application. The syntax and details of the configuration are defined by PAM.

```
public final class LoginContext {
        public LoginContext(String name) { }
        public void login() { }         // two phase process
        public void logout() { }
        public Subject getSubject() { } // get the authenticated Subject
        }

        public interface LoginModule {
        boolean login();   // 1st authentication phase
        boolean commit();  // 2nd authentication phase
        boolean abort();
        boolean logout();
        }
```

Figure 4.1. LoginContext Class and LoginModule Interface

JAAS, like PAM, supports the notion of stacked LoginModules [16]. To guarantee that either all LoginModules succeed or none succeed, the LoginContext performs the authentication steps in two phases. In the first phase, or the 'login' phase, the LoginContext invokes the configured LoginModules (Figure 4.1) and instructs each to attempt the authentication only. If all the necessary LoginModules successfully pass this phase, the LoginContext then enters the second phase and invokes the configured LoginModules again, instructing each to formally 'commit' the authentication process. During this phase each LoginModule associates the relevant authenticated principals and credentials with the subject. If either the first phase or the second phase fails, the LoginContext invokes the configured LoginModules and instructs each to 'abort' the entire authentication attempt. Each LoginModule then cleans up any relevant state they had associated with the authentication attempt.

4.5 Authorization

Once authentication has successfully completed, JAAS provides the ability to enforce access controls upon the principals associated with the authenticated subject. The JAAS principal-based access controls (access controls based on who runs code) supplement the existing Java 2 codesource-based access controls (access controls based on where code came from and who signed it) [9].

4.5.1 Principal-Based Access Control

Services typically implement the access control model of security, which defines a set of protected resources, as well as the conditions under which named principals may access those resources. JAAS also follows this model, and defines a security policy to specify what resources are accessible to authorized principals. The JAAS policy extends the existing default Java 2 security policy, and in fact, the two policies, together, form a single logical access control policy for the entire Java runtime.

```
// Java 2 codesource-based policy
        grant Codebase "http://foo.com", Signedby "foo" {
        permission java.io.FilePermission "/cdrom/-", "read";
        }
```

Figure 4.2. Codesource-Based Policy Entry

Figure 4.2 depicts an example codesource-based policy entry currently supported by the default policy provided with Java 2. This entry grants code loaded from 'foo.com', and signed by 'foo', permission to read all files in the 'cdrom' directory and its subdirectories. Since no principal information is included with this policy entry, the code will always be able to read files from the 'cdrom' directory, regardless of who executes it.

```
// JAAS principal-based policy
        grant Codebase "http://bar.com, Signedby "bar",
         Principal bar.Principal "duke" {
        permission java.io.FilePermission "/cdrom/duke/-", "read";
        }
```

Figure 4.3. Principal-Based Policy Entry

Figure 4.3 depicts an example principal-based policy entry supported by JAAS. This example entry grants code loaded from 'bar.com', signed by 'bar', and executed by 'duke', permission to read only those files located in the '/cdrom/duke' directory. To be executed by 'duke', the subject affiliated with the current access control context must have an associated principal of class, 'bar.Principal', whose 'getName' method returns, 'duke'. If the code from 'bar.com', signed by 'bar', was not executed by 'duke', or if the code was executed by any principal other than 'duke', then it would not be granted the FilePermission. Also if the JAAS policy entry did not specify the Codebase or Signedby information, then the entry's FilePermission would be granted to any code running as 'duke'.

```
// an administrator role can access user passwords
        grant Principal foo.Role "administrator" {
        permission java.io.FilePermission "/passwords/-", "read, write";
        }

        // a basketball team (group) can read its directory
        grant Principal foo.Team "SlamDunk" {
        permission java.io.FilePermission "/teams/SlamDunk/-", "read";
        }
```

Figure 4.4. Role-Based and Group-Based Policy Entries

JAAS treats roles and groups simply as named principals. Therefore access control can be imposed upon roles and groups just as they are with any other type of principal. See Figure 4.4.

For flexibility, the JAAS policy also permits the Principal class specified in a grant entry to be a PrincipalComparator (the class implements the PrincipalComparator interface). The permissions for such entries are granted to any subject that the PrincipalComparator implies.

```
public interface PrincipalComparator {
        boolean implies(Subject subject);
    }

    // regular users can access a temporary working directory
    grant Principal bar.Role "user" {
      permission java.io.FilePermission "/tmp/-", "read, write";
    }
```

Figure 4.5. PrincipalComparator Interface and Example Policy Entry

Figure 4.5 demonstrates how PrincipalComparators can be used to support role hierarchies. In this example assume that an administrator role is senior to a user role and, as such, administrators inherit all the permissions granted to regular users. To accommodate this hierarchy, 'bar.Role' must simply implement the PrincipalComparator interface, and its implies method must return, true, if the provided subject has an associated "administrator" role principal. Note that although the JAAS policy supports role hierarchies via the PrincipalComparator interface, administrators are not limited by it. JAAS can accommodate alternative role-based access control mechanisms (such as that deined in), as long as the alternative access controls can be expressed either through the existing Java 2 policy or the new JAAS policy.

4.5.2 Access Control Implementation

The Java 2 runtime enforces access controls via the java.lang.SecurityManager, and is consulted any time untrusted code attempts to perform a sensitive operation (accesses to the local file system, for example). To determine whether the code has sufficient permissions, the SecurityManager implementation delegates responsibility to the java.security.AccessController, which first obtains an image of the current AccessControlContext, and then ensures that the retrieved AccessControlContext contains sufficient permissions for the operation to be permitted.

JAAS supplements this architecture by providing the method, Subject.doAs, to dynamically associate an authenticated subject with the current AccessControlContext. Hence, as subsequent access control checks are made, the AccessController can base its decisions upon both the executing code itself, and upon the principals associated with the subject. See Figure 4.6.

```
public final class Subject {
        ...
        // associate the subject with the current
        // AccessControlContext and execute the action
        public static Object doAs(Subject s,
                        java.security.PrivilegedAction action) { }
    }
```

Figure 4.6. Subject doAs Method

To illustrate a usage scenario for the doAs method, consider when a service authenticates a remote subject, and then performs some work on behalf of that subject. For security reasons, the server should run in an AccessControlContext bound by the subject's permissions. Using JAAS, the server can ensure this by preparing the work to

be performed as a java.security.PrivilegedAction, and then by invoking the doAs method, providing both the authenticated subject, as well as the prepared PrivilegedAction. The doAs implementation associates the subject with the current AccessControlContext and then executes the action. When security checks occur during execution, the Java 2 SecurityManager queries the JAAS policy, updates the current AccessControlContext with the permissions granted to the subject and the executing codesource, and then performs its regular permission checks.

When the action finally completes, the doAs method simply removes the subject from the current AccessControlContext, and returns the result back to the caller. To associate a subject with the current AccessControlContext, the doAs method uses an internal JAAS implementation of the java.security.DomainCombiner interface. It is through the JAAS DomainCombiner that the existing Java 2 SecurityManager can be instructed to query the JAAS policy without requiring modifications to the SecurityManager itself.

4.6 Scalability

The JAAS principal-based access control policy was intentionally designed to be consistent with the existing codesource-based policy in the Java 2 platform. The default policy implementations provided with both Java 2 and JAAS reside in a local file, and assume that all policy decisions can be defined and made locally. Obviously, this design does not scale beyond small localized environments.

To improve scalability, both the Java 2 and JAAS file-based policy implementations can be replaced with alternative implementations that support delegation. This is achieved by specifying the alternative implementations in the 'java.security' properties file located in the lib/security subdirectory from where the Java runtime environment was installed.

CHAPTER 5

PUBLIC KEY INFRASTRUCTURE (PKI)

5.1 Introduction

Cryptography comes in two flavors: symmetric and assymetric. Symmetric cryptography is when two parties share a secret key that no one else knows. They use this key to encrypt and decrypt messages. If that key were to get loose though, anyone could do the same. Assymmetric cryptography works in key pairs. There is a public and private key. The public key is anywhere and can be used by anyone to encrypt a message that the owner of that public key decrypts with their private key which is safely stored on the smart card. Since symmetric cryptography is generally thousands of times faster than assymetric, a combination of the two is used to achieve the best level of security at the right speed.

5.2 Public Key Cryptography

With the need for information security in today's digital systems growing, cryptography has become one of its critical components. Digital signatures are one of the many uses of cryptography. PKCS#11, also known as Cryptoki, was defined by RSA and is a generic cryptographic token interface.

Public-key cryptography allows one to digitally sign and encrypt information transacted between parties. Public Key Infrastructure (PKI) uses this technology and adds authentication and non-repudiation of the information regarding the parties concerned.

Public Key Cryptography Standards (PKCS) is a suite of protocols and algorithms that are used as an industry standard when implementing public-key cryptography and infrastructure. The fundamentals are based on Key Pairs, Message Digests and Certification.

A key pair consists of a private key and a public key. The private key is never revealed to any party. The public key is made available to the world, or at least the parties concerned with receiving or sending information. In public key algorithms like those from RSA, any data encrypted with the private key can be decrypted only with the public key, and data encrypted with the public key can be decrypted only with the private key. Stronger encryption uses longer keys. For strong encryption, it is "computationally infeasible" to derive the private key given the public key, or vice versa.

Message Digests are hash functions that take in data and generate a statistically unique digest, like a 20 byte number – such that even one bit change in the input data results in a totally different digest. Thus these digests serve as finger-prints of a document. Given a digest and a document, and knowing the hash algorithm, it is easy to verify whether the digest is derived from the document.

Certification is the mechanism by which authenticity is established. A party generates a key pair consisting of the private and public keys. The public key is placed into a certificate request and sent to a certifying authority (CA) like Thawte, IDCertify, VeriSign and so on. The certifying authority (CA) verifies the party's credentials and the purpose of using the keys, through a vetting process, and then certifies the public key they received. That is, the authority issues a certificate, typically called an X.509 digital

certificate that contains the details of the party, the intended use of the certificate and most importantly, the party's public key. This information is then digitally signed by the CA using the CA's private key. The authenticity of the certificate itself can be verified by using the CA's public key, which is made available from the CA's web site, or comes embedded in a browser by default.

In essence, if we trust the CA, then we can trust that the public key in the verified certificate indeed belongs to who ever the CA says it belongs, and therefore if a digital signature on a document is verified using that public key, the information therein was indeed signed by the party mentioned in the certificate. This establishes authenticity, since only the holder of the corresponding private key could have created that digital signature. And trust in the CA is at the core of this process. If a CA is granted a notary or equivalent status, then the certificate and the information signed or encrypted cannot be repudiated and is valid in many courts of law.

5.3 Digital Signatures and Data Encryption

A digital signature is a digital attestation of a document by a party [10]. This is to establish authenticity. A digital signature is an encrypted digest (hash) of the data to be signed.

One essentially creates a digest or hash (using an algorithm like MD5 or SHA1) from the document data and then encrypts this hash with one's private key. The encrypted hash thus becomes a digitally signed finger-print for that document, called a digital signature. This signature can now optionally be attached to the document, along

with one's certificate. Anyone intent on verifying the digital signature would verify the certificate for authenticity first, then take the public key from the certificate and then verify the digital signature. The latter part involves decrypting the digital signature with the public key to reveal the digest or hash value. The document is then hashed using the same algorithm to check whether the digest values match.

A digital signature is typically attached to a document. This can be difficult for certain document types. It is required to embed the signature into the document without changing the document, which is contradictory. Therefore a signing process only works on the information portion of a document, and uses other sections of the format to embed the signature. For example it is possible to embed signatures into a Word document treating the latter as an OLE compound document. One may also store signatures as attributes of such a document. PDF is another format that is amenable to embedding using the DIGSIG API. Multiple signatures may be created and attached to a document. The signatures may be peer level or hierarchical level. Peer level signatures imply that one or more parties have endorsed the document by applying their signatures. Hierarchical signatures imply a work-flow and counter-signing process.

Creation of a digital signature involves using one's private key. In contrast, encryption of information meant for another party uses the other party's public key. Anyone, knowing that party's public key can send encrypted information. Only that party can decrypt the information, using his/her own private key.

5.3.1 Applications

Digitally signed documents provide the authenticity of paper documents, and convenience of electronic documents. This has made digitally signed documents a great success. Some of the applications of digitally signed documents are in “legal document” centric industries like law firms, banking, and stock broking. It also has high potential in government sector, where powerful document management software can make the system more efficient and fast.

Digital signatures also make transactions happen much faster. For example the ownership of a cargo carried by an oil tanker from the Persian Gulf to New York may change hands 5 to 10 times by the time the tanker reaches New York. With digital signatures applied to contracts on e-Marketplaces, they may change hands maybe 50 times or more, greatly speeding up the trading process.

5.4 PKCS #11

5.4.1 Introduction

The PKCS #11 standard describes a programming interface that can work with cryptographic tokens and devices such as smart cards and PCMCIA cards [11]. Through this interface one may initialize such devices, create key pairs, store certificates, digitally sign data etc. A PKCS#11 interface is typically implemented by a software driver that works in conjunction with the hardware reader and the token, and translates interface calls to token-specific controls. This well-defined interface allows even a browser to interact with such a token and make use of its capabilities.

Portable computing devices such as smart cards, PCMCIA cards, and smart diskettes are ideal tools for implementing public-key cryptography, as they provide a way to store the private-key component of a public-key/private-key pair securely, under the control of a single user. With such a device, a cryptographic application utilizes the device to perform the operations, with sensitive information such as private keys never being revealed.

5.4.2 Overview

- General model

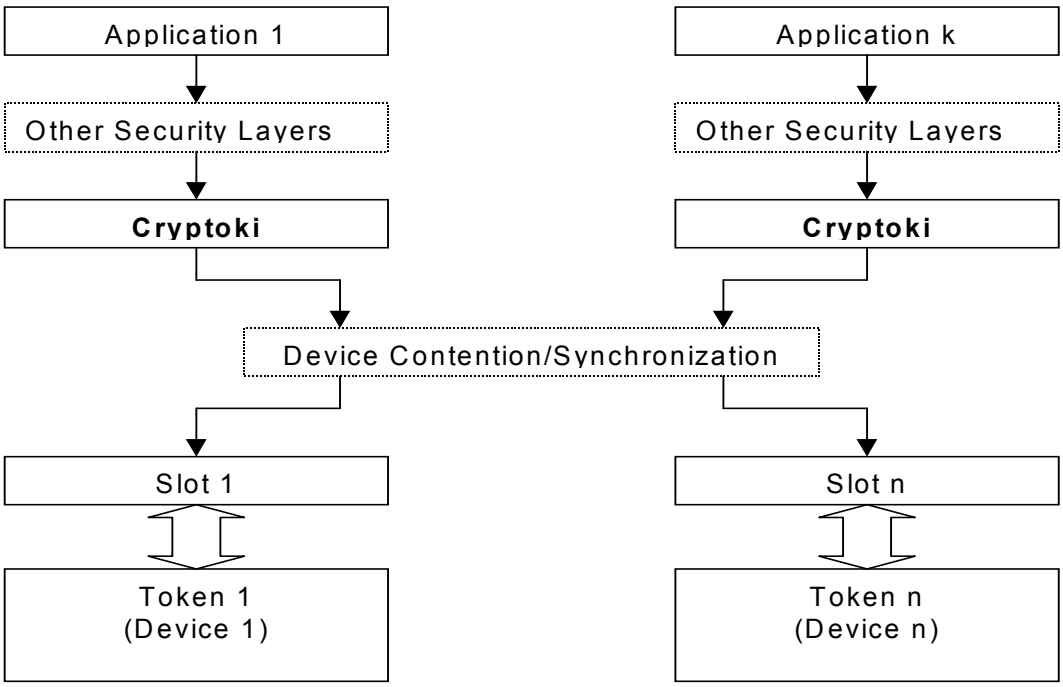


Figure 5.1. General Cryptoki Model

Cryptoki's general model is illustrated in Figure 5.1. The model begins with one or more applications that need to perform certain cryptographic operations, and ends with one or more cryptographic devices, on which some or all of the operations are actually performed. A user may or may not be associated with an application.

Cryptoki provides an interface to one or more cryptographic devices that are active in the system through a number of "slots". Each slot, which corresponds to a physical reader or other device interface, may contain a token. A token is typically "present in the slot" when a cryptographic device is present in the reader. Of course, since Cryptoki provides a logical view of slots and tokens, there may be other physical interpretations. It is possible that multiple slots may share the same physical reader. The point is that a system has some number of slots, and applications can connect to tokens in any or all of those slots.

An application may be linked to Cryptoki directly; alternatively, Cryptoki can be a so-called "shared" library (or dynamic link library), in which case the application would link the library dynamically. Shared libraries are fairly straightforward to produce in operating systems such as Microsoft Windows and OS/2, and can be achieved without too much difficulty in UNIX and DOS systems.

The kinds of devices and capabilities supported depend on the particular Cryptoki library. This standard specifies only the interface to the library, not its features. In particular, not all libraries support all the mechanisms (algorithms) defined in this interface (since not all tokens are expected to support all the mechanisms), and libraries

are likely to support only a subset of all the kinds of cryptographic devices that are available.

- PKCS#11 Object Model

As seen in Figure 5.2, this is a simple object model but powerful enough for the operations expected from a token. The private key is used for generating a digital signature and the public key (which is also a part of the certificate) is used for verification of the digital signature. Secret key or PIN is used for managing keys and certificates from the store. A token can typically store a number of such objects.

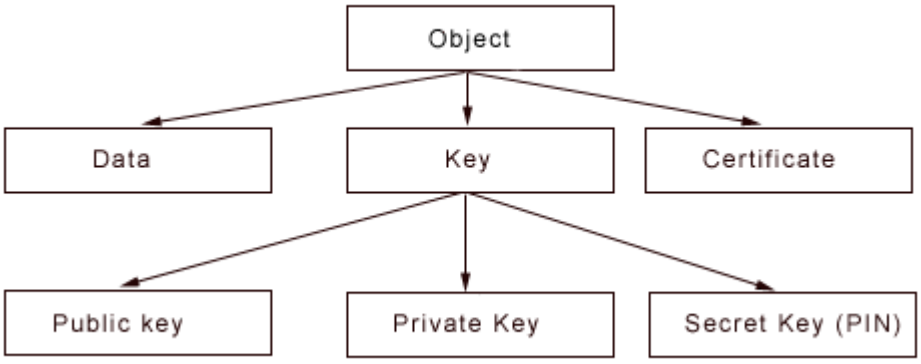


Figure 5.2. Object Model

5.5 PKCS #12

5.5.1 Introduction

PCKS #12 standards describe transfer syntax for personal identity information, including private keys, certificates, miscellaneous secrets, and extensions [12]. Entities like machines, applications, browsers, Internet kiosks, and so on, that support this

standard will allow a user to import, export, and exercise a single set of personal identity information.

This standard supports direct transfer of personal information under several privacy and integrity modes. The most secure of the privacy and integrity modes require the source and destination platforms to have trusted public/private key pairs usable for digital signatures and encryption, respectively. The standard also supports lower security, password-based privacy and integrity modes for those cases where trusted public/private key pairs are not available.

This standard is amenable to both software and hardware implementations. Hardware implementations offer physical security in tamper-resistant tokens such as smart cards and PCMCIA devices.

5.5.2 Overview

- Exchange modes

There are four combinations of privacy modes and integrity modes. Encryption is used by the privacy mode to protect personal information from exposure, and personal information is protected from tampering by integrity mode. Without protection from tampering, an adversary could substitute invalid information for the user's personal information without the user being aware of the substitution.

The following are the privacy modes:

- *Public-key privacy mode*: Personal information is enveloped on the source platform using a trusted encryption public key of a known destination platform. The envelope is opened with the corresponding private key.
- *Password privacy mode*: Personal information is encrypted with a symmetric key derived from a user name and a privacy password. If password integrity mode is used as well, the privacy password and the integrity password may or may not be the same.

The following are the integrity modes:

- *Public-key integrity mode*: Integrity is guaranteed through a digital signature on the contents of the Protocol Data Unit (PDU), which is produced using the source platform's private signature key. The signature is verified on the destination platform by using the corresponding public key.
- *Password integrity mode*: Integrity is guaranteed through a message authentication code (MAC) derived from a secret integrity password. If password privacy mode is used as well, the privacy password and the integrity password may or may not be the same.

➢ Mode choice policies

All combinations of the privacy and integrity modes are permitted in this standard. Good security policy suggests that certain practices be avoided, e.g., it can be unwise to transport private keys without physical protection when using password privacy mode or when using public-key privacy mode with weak symmetric encryption.

Unfortunately, weak symmetric encryption may be required for products exported from certain countries under applicable export regulations.

In general, the public key modes for both privacy and integrity are preferable to the password modes (from a security viewpoint). However, it is not always possible to use the public key modes. For example, it may not be known at export time what the destination platform is; if this is the case, then the use of the public-key privacy mode is precluded.

- Trusted public keys

Asymmetric key pairs may be used in this standard in two ways: public-key privacy mode and public-key integrity mode. For public-key privacy mode, an encryption key pair is required; for public-key integrity mode, a signature key pair is required.

It may be appropriate for the keys to be platform-specific keys dedicated solely for the purpose of transporting a user's personal information. Whether or not that is the case, though, the keys should not be confused with the user's personal keys that the user wishes to transport from one platform to another (these latter keys are stored within the PDU).

For public-key privacy mode, the private key from the encryption key pair is kept on the destination platform, where it is ultimately used to open a private envelope. For public-key integrity mode, the private key from the signature pair is kept on the source platform, where it is used to sign personal information.

For both uses of public/private key pairs, the public key from the key pair must be transported to the other platform such that it is trusted to have originated at the correct platform. Judging whether or not a public key is trusted in this sense must ultimately be left to the user. There are a variety of methods for ensuring that a public key is trusted.

- The AuthenticatedSafe

Each compliant platform shall be able to import and export AuthenticatedSafe PDUs wrapped in PFX PDUs. For integrity, the AuthenticatedSafe is either signed (if public-key integrity mode is used) or MACed (if password integrity mode is used) to produce a PFX PDU. If the AuthenticatedSafe is signed, then it is accompanied by a digital signature, which was produced on the source platform with a private signature key, corresponding to a trusted public signature key. Public signature key must accompany the PFX to the destination platform, where the user can verify the trust in the key and can verify the signature on the AuthenticatedSafe. If the AuthenticatedSafe is MACed, then it is accompanied by a Message Authentication Code computed from a secret integrity password; salt bits; an iteration count and the contents of the AuthenticatedSafe.

The AuthenticatedSafe itself consists of a sequence of ContentInfo values, some of which may consist of plaintext (data), and others which may either be enveloped (if public-key privacy mode is used) or encrypted (if password privacy mode is used). If the contents are enveloped, then they are encrypted with a symmetric cipher under a freshly generated key, which is in turn encrypted with RSA asymmetric encryption. The RSA public key used to encrypt the symmetric key corresponds to an RSA private key, on the

destination platform. This public key needs to be trusted by the user when it is used at export time. If the contents are encrypted, then they are encrypted with a symmetric cipher under a key derived from a secret privacy password, salt bits and an iteration counter.

Each ContentInfo contains an arbitrary collection of private keys, PKCS #8 shrouded private keys, certificates, CRLs, or opaque data objects, at the user's discretion, stored in values of type SafeContents.

The reason for the unencrypted option is that some governments restrict certain uses of cryptography. Having several parts in an AuthenticatedSafe keeps implementers' options open. For example, it may be the case that strong cryptography can be used to make PKCS #8-shrouded keys, but then these shrouded keys should not be further encrypted, because super-encryption can limit a product's exportability.

Around the AuthenticatedSafe is the integrity-mode wrapper, which protects the entire contents of the AuthenticatedSafe This is the reverse of the wrapping order in many protocols, in which privacy is the outermost protection. This latter, more common wrapping order avoids signatures on encrypted data, which are undesirable under certain circumstances and it is therefore preferable to protect the integrity of as much information as possible.

5.6 PKCS# 15

5.6.1 Introduction

PKCS #15 establishes a standard that enable users in to use cryptographic tokens to identify themselves to multiple, standards-aware applications, regardless of the application's cryptoki (or other token interface) provider [13].

Cryptographic tokens, such as Integrated Circuit Cards (or IC cards) are intrinsically secure computing platforms ideally suited to providing enhanced security and privacy functionality to applications. They can handle authentication information such as digital certificates and capabilities, authorizations and cryptographic keys. Furthermore, they are capable of providing secure storage and computational facilities for sensitive information such as:

- Private keys and key fragments;
- Account numbers and stored value;
- Passwords and shared secrets; and
- Authorizations and permissions.

At the same time, many of these tokens provide an isolated processing facility capable of using this information without exposing it within the host environment where it is at potential risk from hostile code (viruses, Trojan horses, and so on). This becomes critically important for certain operations such as:

- Generation of digital signatures, using private keys, for personal identification;
- Network authentication based on shared secrets;
- Maintenance of electronic representations of value; and

- Portable permissions for use in off-line situations.

Unfortunately, the use of these tokens for authentication and authorization purposes has been hampered by the lack of interoperability at several levels. First, the industry lacks standards for storing a common format of digital credentials (keys, certificates, etc.) on them. This has made it difficult to create applications that can work with credentials from a variety of technology providers. Attempts to solve this problem in the application domain invariably increase costs for both development and maintenance. They also create a significant problem for the end-user since credentials are tied to a particular application running against a particular application-programming interface to a particular hardware configuration.

Second, mechanisms to allow multiple applications to effectively share digital credentials have not yet reached maturity. While this problem is not unique to cryptographic cards - it is already apparent in the use of certificates with World Wide Web browsers, for example - the limited room on many cards together with the consumer expectation of universal acceptance will force credential sharing on credential providers. Without agreed-upon standards for credential sharing, acceptance and use of them both by application developers and by consumers will be limited.

To optimize the benefit to both the industry and end-users, it is important that solutions to these issues be developed in a manner that supports a variety of operating environments, application programming interfaces, and a broad base of applications. Only through this approach can the needs of constituencies be supported and the

development of credentials-activated applications encouraged, as a cost-effective solution to meeting requirements in a very diverse set of markets.

The objectives of this document are therefore to:

- Enable interoperability among components running on various platforms (platform neutral);
- Enable applications to take advantage of products and components from multiple manufacturers (vendor neutral);
- Enable the use of advances in technology without rewriting application-level software (application neutral); and
- Maintain consistency with existing, related standards while expanding upon them only where necessary and practical.

As a practical example, the holder of an IC card containing a digital certificate should be able to present the card to any application running on any host and successfully use the card to present the contained certificate to the application.

As a first step to achieve these objectives, this document specifies a file and directory format for storing security-related information on cryptographic tokens. It has the following characteristics:

- Dynamic structure enables implementations on a wide variety of media, including stored value cards;
- Allows multiple applications to reside on the card (even multiple eid applications);
- Supports storage of any type of objects (keys, certificates and data); and

- Support for multiple pins whenever the token supports it.

➢ Information access model

The PKCS #15 token information may be read when a token is presented containing this information, and is used by a PKCS #15 interpreter which is part of the software environment, e.g. as shown in Figure 5.3.

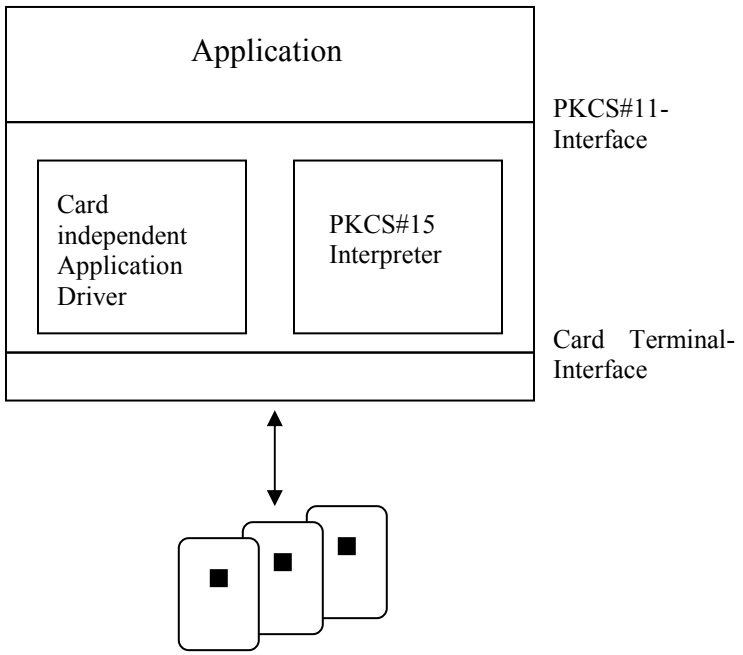


Figure 5.3. Embedding of a PKCS #15 interpreter (example)

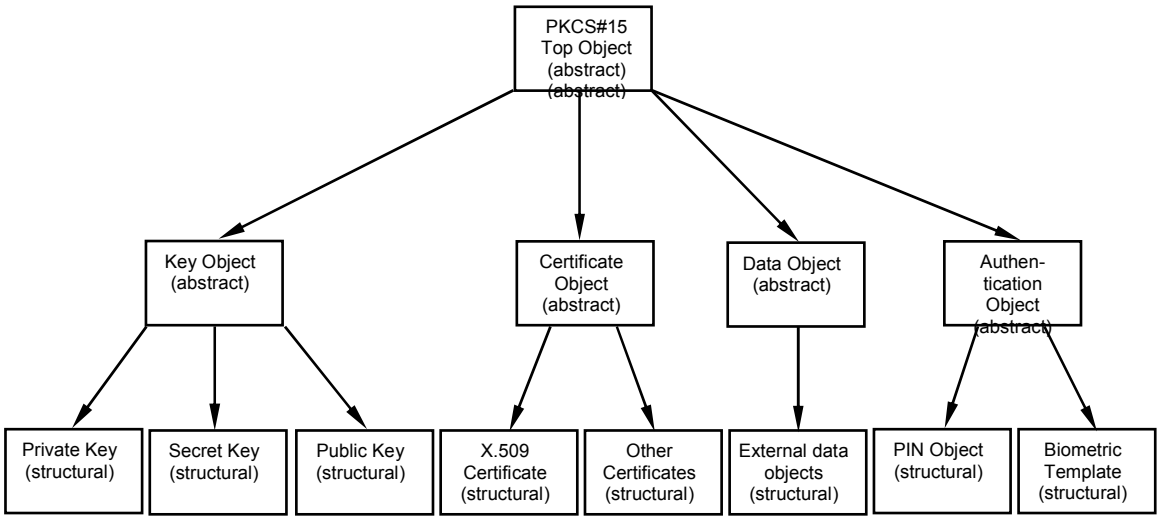
5.6.2 Overview

➢ Object classes

There are four general classes of objects: Keys, Certificates, Authentication Objects and Data Objects. All these object classes have sub-classes, e.g. Private Keys, Secret Keys and Public Keys, whose instantiations become objects actually stored on cards. Figure 5.4 shows object hierarchy.

- Attribute types

All objects have a number of attributes. Objects "inherits" attribute types from their parent classes (in particular, every object inherit attributes from the abstract PKCS #15 "Common" or "Top" object).



NOTE – instances of abstract object classes does not exist on cards

Figure 5.4. PKCS #15 Object hierarchy

- Access methods

Objects can be private, meaning that they are protected against unauthorized access, or public. In the IC card case, access (read, write, etc) to private objects is defined by Authentication Objects (which also includes Authentication Procedures). Conditional access (from a cardholder's perspective) is achieved with knowledge-based or biometric user information. In other cases, such as when PKCS #15 is implemented in software, private objects may be protected against unauthorized access by cryptographic means.

Public objects are not protected from read-access. Whether they are protected against modifications or not depends on the particular implementation.

CHAPTER 6

OPEN CA

6.1 Introduction

Public key infrastructures are one of the most widely accepted musts of the future. The problem is that more and more applications can be secured with such crude things like certificates and keys but it is really difficult to setup PKIs and it is really expensive too because flexible trust center software for Unix is expensive. The goal of OpenCA is the production of an open source trust center system to support the community with a good, inexpensive and future-proof solution for their base infrastructure.

The idea consists of three major parts - a Perl web interface, an OpenSSL backend for the cryptographic operation and a database [14]. Nearly all operations can be performed via some web interfaces. The cryptographic backend is OpenSSL. The aim is to provide the organizational infrastructure for a PKI. Databases store all the needed information about the user's crypto objects like certificate signing requests, certificates, certificate revocation requests and CRLs.

OpenCA support the following things:

- Public interface
- LDAP interface
- RA interface
- CA interface
- SCEP

- OCSP
- IP-filters for interfaces
- Passphrase based login
- Certificate based login (incl. smatcards)
- Role Based Access Control
- flexible certifcate subjects
- flexible certificate extensions
- PIN based revocation
- digital signature based revocation
- CRL issuing
- Warnings for expiring certificates
- support for nearly every (graphical) browser

OpenCA is designed for a distributed infrastructure. It cannot only handle an offline CA and an online RA. We can build a hierarchy with three or more levels. The goal is a maximum flexibility to support big organizations like universities, grids and global companies. OpenCA is not only a small solution for small and medium research facilities.

6.2 Basic Hierarchy

The basic idea of every X.509 PKI (Public Key Infrastructure) is a strong hierarchical organization. This results in a tree of databases if we try to create a distributed PKI architecture.

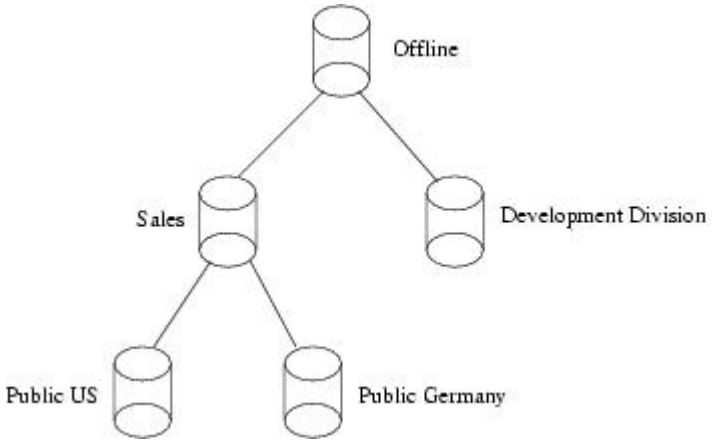


Figure 6.1. Distributed PKI Architecture

The data exchange between such isolated databases (Figure 6.1) can be handled automatically if we use a distributed database system but in the sense of OpenCA such a distributed database system is only one database in our tree. If we really have an isolated database (e.g. for an Offline CA) then we must have the technology for the data exchange and the management of the complete node in the hierarchy. This management functionality is bundled in an interface called node or node management. Hence the design of OpenCA looks like Figure 6.2.

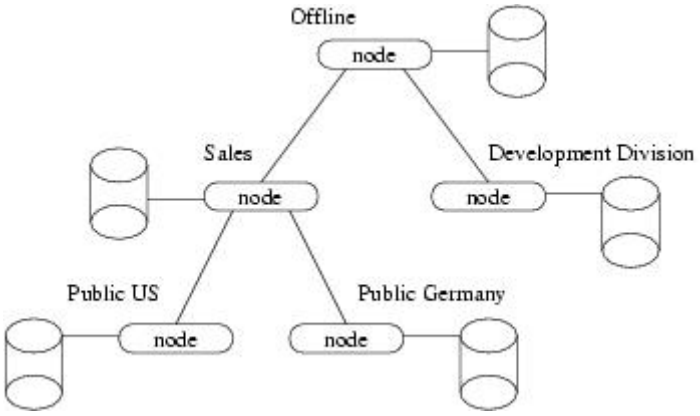


Figure 6.2. OpenCA Design

Normally every server in the infrastructure of the trust center has its own database for security reasons. This hierarchy is the backbone of the trust center.

6.3 Interfaces

After we know the basic infrastructure of OpenCA we think about things like CA, RA, LDAP and a public interface which is sometimes called web-gateway. OpenCA supports all these software components via special web interfaces.

6.3.1 Node

This interface manages the database and handles all the export and import functionalities. The database can be initialized what means that OpenCA can create all the tables but OpenCA cannot create the database itself because this differs for every vendor. So we need a database with the appropriate access rights and a new database. The interface includes some functions for the backup and recovery of such a node but we you MUST have a separate backup of the CA's private key and certificate. There is no default mechanism in OpenCA to backup the private key.

The export and import will be handled by this interface too. We can configure different rules for the synchronization with nodes on a higher and a lower level of the hierarchy. This includes the configuration of the objects and status which can be exchanged. The configured filters avoid status injections from lower levels of the hierarchy.

6.3.2 CA

The CA interface has all the functions which you need to create certificates and Certificate Revocation Lists (CRLs). The CA also includes all the functions which you

can use to change the configuration via a web interface. It is not possible to change the configuration via another web interface.

The CA is the home of the batch processors too. OpenCA includes some powerful batch processors for creating certificates. These batch processors can be used for automatic certificate creation from various Enterprise Resource Planning (ERP) systems (e.g. SAP, HIS, NIS or /etc/passwd).

6.3.3 RA

OpenCA's RA is able to handle all kinds of requests. This include things like editing requests, approving requests, creating private keys with smart cards, delete wrong requests and email users.

6.3.4 LDAP

The LDAP interface was implemented to separate the LDAP management completely from the rest of the software. This is necessary because there are many functions which are really specific for LDAP admins, with only a few users needing these features.

6.3.5 Pub

The Public interface includes all the small things which the users need. This is only a small list and perhaps it is incomplete

- Generates CSRs (certificate signing request) for Microsoft Internet Explorer

- Generates CSRs for Mozilla 1.1+ and Netscape Communicator and Navigator
- Generates client independent requests and private keys (e.g. for KDE's konqueror or server administrators who don't know how to create a private key and request)
- Receives PEM-formatted PKCS\#10 requests from servers
- Enrolls certificates
- Enrolls CRLs
- Supports two different methods revocation
- Search certificates
- Tests user certificates in browsers (Microsoft IE and Netscape Navigator 4.7x)

If we want to design a powerful trust center then we must have a concept about how we want to organize our work flow. We can see an example in the Figure 6.3.

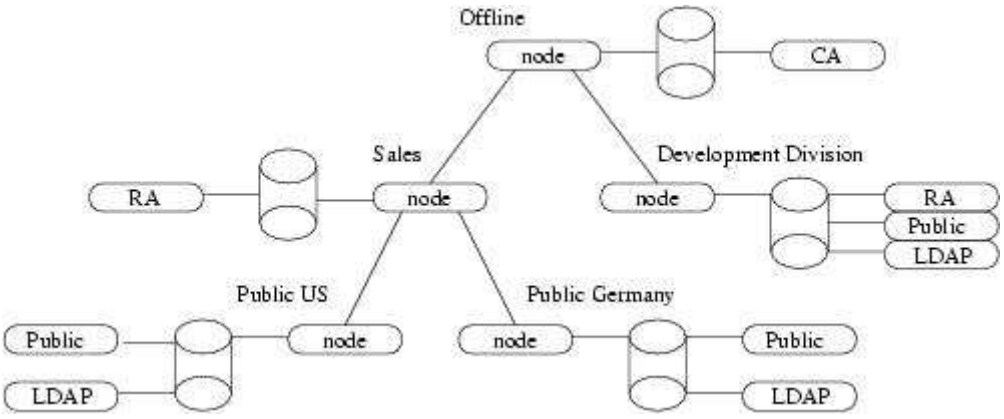


Figure 6.3. OpenCA Workflow

CHAPTER 7

OPENCARD FRAMEWORK AND SMART CARD

7.1 Introduction

In order to use a smart card, we need to be able to read the card and communicate with it using an application. OpenCard provides a framework for this by defining interfaces that must be implemented. The OpenCard framework defines several of these interfaces. Once these interfaces are implemented, we can use other services in the upper layers of the API. For example, with a properly interfaced reader, OpenCard can start a Java card agent whenever the card is inserted. The card agent can then communicate with applications on the smart card via the card terminal in the context of a session.

While it's not necessary to use OpenCard in creating 100% pure Java smart card applications, without it developers are forced to use home-grown interfaces to smart cards. OpenCard also provides developers with an interface to PC/SC (a smart card application interface developed by Microsoft and others for communicating with smart cards from Win32-based platforms for PCs) for use of existing devices on Win32 platforms.

Smart card applications consist of a card-external application interacting with a card-resident component. The card-external application comprises the program code running on some computing platform such as a personal computer, a network computer, an ATM (automatic-teller machine), or a personal digital assistant. The card-resident component comprises data and functions in the smart card itself.

Interactions with the smart card occur by exchanging pairs of APDUs (application protocol data units) and are initiated by the external application. Communication with the smart card is accomplished via the card reader, an I/O device attached to the computing platform into which the smart card is inserted. The application sends a CommandAPDU to the smart card by handing it to the card reader's device driver, which forwards it to the smart card. In return, the card sends back a ResponseAPDU, which the device driver hands back to the application.

A smart card's functions are determined by the set of CommandAPDUs that it understands. Although standards for smart cards do exist, the functionality may vary significantly. In other words, depending on the card vendor and/or the type of card, different functionalities are offered and the exact definition of the set of Command- and ResponseAPDUs may differ.

Similarly, there is a broad range of card readers on the market with widely differing functionalities. Very simple readers merely provide basic reader functionality offering a single slot to insert the card. More sophisticated readers offer multiple slots or include a PIN3 pad and a display that can be used to perform cardholder verification. Card readers can attach to different I/O ports including serial port and PC Card slot. There are even versions available that can be inserted into the floppy drive. They come with proper driver software for a selection of operating systems. There is no single API to access the card reader driver.

Given these technicalities, development of smart card applications has been much of an art in the past and was confined to a relatively small number of specialized

programmers. In addition, the dependencies on the make of the card and the card reader have usually prevented application developers from deploying their OpenCard Architecture

OpenCard provides architecture for developing applications in Java that utilize smart cards or other ISO 7816-compliant devices on different target platforms such as Windows, network computers, Unix workstations, Webtops, set tops, and so on. The OpenCard Framework provides an application programming interface (API), which allows us to register cards, look for cards in readers, and optionally have Java agents start up when cards are inserted in the reader. The architecture of OpenCard is depicted in Figure 7.1.

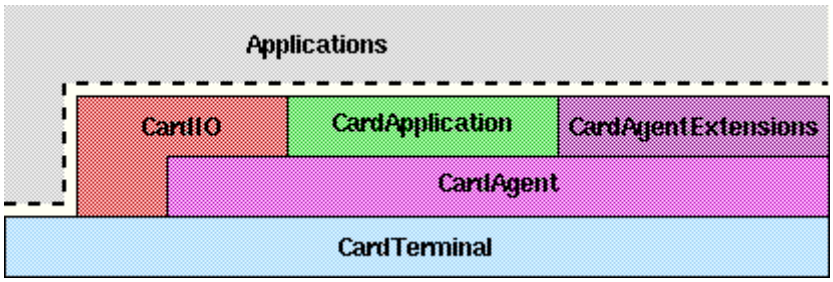


Figure 7.1. OpenCard Framework architecture

The architecture of the OpenCard Framework is made up of the CardTerminal, the CardAgent, the Agents and/or applications that interact with these components. OpenCard consists of four Java packages with the prefix opencard.

7.2 OpenCard Framework

Developers are normally concerned with implementing both parts of a smart card application, the card-resident component and the card-external program. OCF helps

developers primarily in developing the card-external program code. They can then program against high-level APIs that let make the functionality and information contained in the smart card fully accessible. Having OCF sit in between the application and the smart card hides the complexity of interacting with the smart card. At the same time, the high-level of abstraction offered by the OPENCARD FRAMEWORK [15] achieves the desired transparency with regard to dependencies on the particular type of smart card and/or card reader used in a particular setting. By virtue of its framework nature, OPENCARD FRAMEWORK adapts its capabilities at run-time to match the characteristics of a particular card reader device and/or the particular smart card inserted into the reader. For programmers, this means that they can concentrate on the application logic and need not be concerned with the intricacies of dealing with a particular reader or card.

In order to adapt itself to a particular situation, OCF relies on the availability of adequate JAVA components that can be plugged into the framework to address a particular card reader and card inserted. But don't worry, these components are typically developed by card reader manufacturers and card-chip manufacturers and are not something which application programmer have to implement. All they have to do is to make sure that the card reader and card type they choose to deploy the application are supported by OCF-compliant components. An up-to-date list of the devices supported can be found at the OCF web site. OCF will not provide support in developing the card-resident part of the application. Normally, smart card vendors offer their cards together

with a development toolkit that supports the development of the card-resident application component.

For conventional smart cards, the toolkit may contain tools supporting all or a subset of the following tasks:

- Layout Definition

This is the process of generating an EEPROM image from a high-level definition of the EEPROM layout. Most smart card applications maintain information in the smart card. This information is typically kept in EEPROM-based files on the card's file system. Layout definition is about identifying the information items that should go in the card and defining an appropriate file structure in the card. The latter includes specifying the type of file (transparent, record-oriented, cyclic), file names and/or identifiers, access conditions, initial data, etc.

- Card initialization

This is the process of writing initialization data to the smart card EEPROM (it could be compared to formatting a disk). The EEPROM image generated during the layout definition is transferred to the smart card. Depending on volume, this can be supported by special types of card readers that achieve a high throughput.

- Card personalization

This is the process of writing cardholder-specific data to the smart card. After initialization, the smart card EEPROM reflects the basic file system structure as defined in the layout definition and may contain some initial data that is constant across all cards, such as meta-information about the file system structure, cryptographic key information,

etc. During personalization, the information peculiar to an individual cardholder is written to the card prior to issuing it.

7.2.1 The Terminal Package

The packages opencard.application and opencard.io provide the high-level API used by the application developer. The services needed by the high-level API are carried out by classes in the opencard.agent and opencard.terminal packages. The opencard.agent package abstracts the functionality of the smart card through the CardAgent. Package opencard.terminal abstracts the card terminals (also known as card readers). Understanding the structure of the opencard.terminal package is required to understand the sample implementations of card terminals provided in this article.

A card terminal abstracts the device that is used in a computer system to communicate with a smart card. The opencard.terminal package contains classes to represent the card-terminal hardware, to interact with the user, and to manage card-terminal resources. Not all readers have these abilities. When implementing a reader that doesn't have keyboard entry, we will use the UserInteractionHandler.

7.2.2 Card Terminal Representation

Each card terminal is represented by an instance of class CardTerminal that defines the abstract OpenCard-compliant card terminal. A card terminal may have one or more slots for smart cards and optionally a display and a keyboard or PIN pad. The slots of a card terminal are represented by instances of the abstract class Slot, which offers

methods to wait for a card to be inserted, to communicate with the card, and to eject it (if possible).

7.2.3 User Interaction

Using a smart card requires interaction with the user -- for card-holder verification. The interface UserInteraction provides for this functionality. It provides methods to write a message onto the display and receive input from the user. Card terminals that do not support all user interaction features can make use of the UserInteractionHandler, which implements a UserInteraction as a graphical user interface based on the abstract windowing toolkit (AWT).

7.2.4 Resource Management

Cards and card readers require resource management so that agents can be granted the level of access control they require. Resource management provides for the sharing of card terminals and the cards inserted in them among the agents in the system. For example, say you are using your smart card to sign a document at the same time that a high-priority mail message comes in that needs to be decoded using your smart card. Resource management arbitrates the access to the CardTerminal and the correct port.

The resource management for card terminals is achieved by the CardTerminalRegistry class of OpenCard. There is only one instance of CardTerminalRegistry: the system-wide card terminal registry. The system-wide card terminal registry keeps track of the card terminals installed in the system. The card

terminal registry can be configured from properties upon system start up or dynamically through register and unregister methods to dynamically add or remove card terminals from the registry.

During the registration of a card terminal, a CardTerminalFactory is needed to create an instance of the corresponding implementation class for the card terminal. The card terminal factory uses the type name and the connector type of the card terminal to determine the CardTerminal class to create. The concept of a card terminal factory allows a card terminal manufacturer to define a mapping between user-friendly type names and the class name.

7.3 Smart Card

A smart card is a card that is embedded with either a microprocessor and a memory chip or only a memory chip with non-programmable logic [17]. The microprocessor card can add, delete, and otherwise manipulate information on the card, while a memory-chip card (for example, pre-paid phone cards) can only undertake a pre-defined operation.

Smart cards, unlike magnetic stripe cards, can carry all necessary functions and information on the card. Therefore, they do not require access to remote databases at the time of the transaction.

Smart cards are small, tamper-resistant devices providing users with convenient storage and processing capability. The smart cards are suitable for cryptographic implementations because they contain many security features that enable the protection

of sensitive cryptographic data and provide for a secure processing environment. The protection of the private key is critical for digital signatures. This key must never be revealed. The smart card protects the private key, and many consider the smart card an ideal cryptographic token. The private key is generated by the smart card and never leaves the smart card providing high level of security.

Today, there are three categories of smart cards, all of which are evolving rapidly into new markets and applications:

- Integrated Circuit (IC) Microprocessor Cards
- Integrated Circuit (IC) Memory Cards.
- Optical Memory Cards.

7.3.1 Card Acceptance Device

A smart card is inserted into card acceptance device (CAD) which may connect to another computer. Card acceptance device can be of two types: readers and terminals. Readers are connected to serial, parallel or USB ports of the computer through which it communicates. A reader has a slot in which card is placed or it can receive data carried through electromagnetic fields from the contactless card.

7.3.2 Smart Card Operating System

The Smart operating system have a little resemble to our desktop operating systems. Smart Card OS support a collection of instructions on which user applications can build. ISO 7816-4 standardize a wide range of instructions in a format of APDUs. A

smart card operating system may support some or all of APDUs as well as manufacturers extensions and additions. ISO 7816-4 are largely file system-oriented commands, such as file selection and file access commands. A user application often is a data-file that stores application-specific information. The semantics and applications to access the information to access the application data file are implemented by operating system. Therefore, the separation between the operating system and applications are not well defined. This file-centric operating system is established in smart cards that are available today. However newer operating systems which support a better system-layer separation and downloading of custom application code are becoming more popular today. Java Card Technology is one technology in the new trend.

7.3.3 Smart Card File System

Smart Cards defined in ISO 7816-4 can have a hierarchical file system structure. Each file is specified by either a 2-byte identifier or a symbolic name up to 16 bytes. To perform file operations a file must be selected (like opening file). Access to file is controlled by access conditions which can be specified differently by for read and write access. Card operating system file are divided into three types [23]:

Master File (MF): It is the root of the file system. The master file can contain dedicated and elementary files. There will be only one MF in one card.

Dedicated File (DF): A DF is a smart card directory file that holds other dedicated files and other elementary files. A master File is a special type of DF.

Elementary File (EF): An EF is a data file it can't contain other files. Based on file structure there are four types of Elementary Files

- *Transparent File*- It is structured as sequence of data bytes; where as other three are structured as a sequence of individually identifiable records.
- *Linear Fixed File*- It has records of fix size.
- *Linear Variable*- It has records of variable sizes.
- *Cyclic File*- It has fixed size record organize in a Ring

CHAPTER 8

MUSCLE CARD CRYPTOGRAPHIC TOKEN FRAMEWORK

8.1 Introduction

Smartcards require large amounts of complex middleware that communicates with the card and exports the card's functionality to the host. These cards typically vary from release to release so the middleware generally is in constant change. Currently each card must have its own CSP (crypto/card service provider) on the host creating large support problems and security trust well beyond most OS vendor's preferences.

Using MUSCLE applet approach [18], it is required that only one host CSP be written for the middleware, thus reducing the time spent migrating to new card releases and vastly reducing the number of CSP's on the host. The MUSCLE applet has to be loaded on the card with a static application identifier (AID) and the host based CSP will communicate to the card through this applet. The Java Card API's support a wide array of cryptographic capability including both symmetric and asymmetric functions, random number generation, key generation/management, and PIN management.

The Applet is capable of generating cryptographic keys on the card, and allows external keys to be inserted onto the card. These keys can be used in cryptographic operations, after proper user (or host application) authentication. The Applet is capable of handling generic objects. An object is a sequence of bytes whose meaning is determined by the application. The Applet allows a host application to read and/or modify objects' contents, after proper user (or host application) authentication.

8.2 MUSCLE Architecture

MUSCLE [19] is a project to coordinate the development of smart cards and applications under Linux. The purpose is to develop a set of compliant drivers, API's, and a resource manager for various smart cards and readers for the GNU environment.

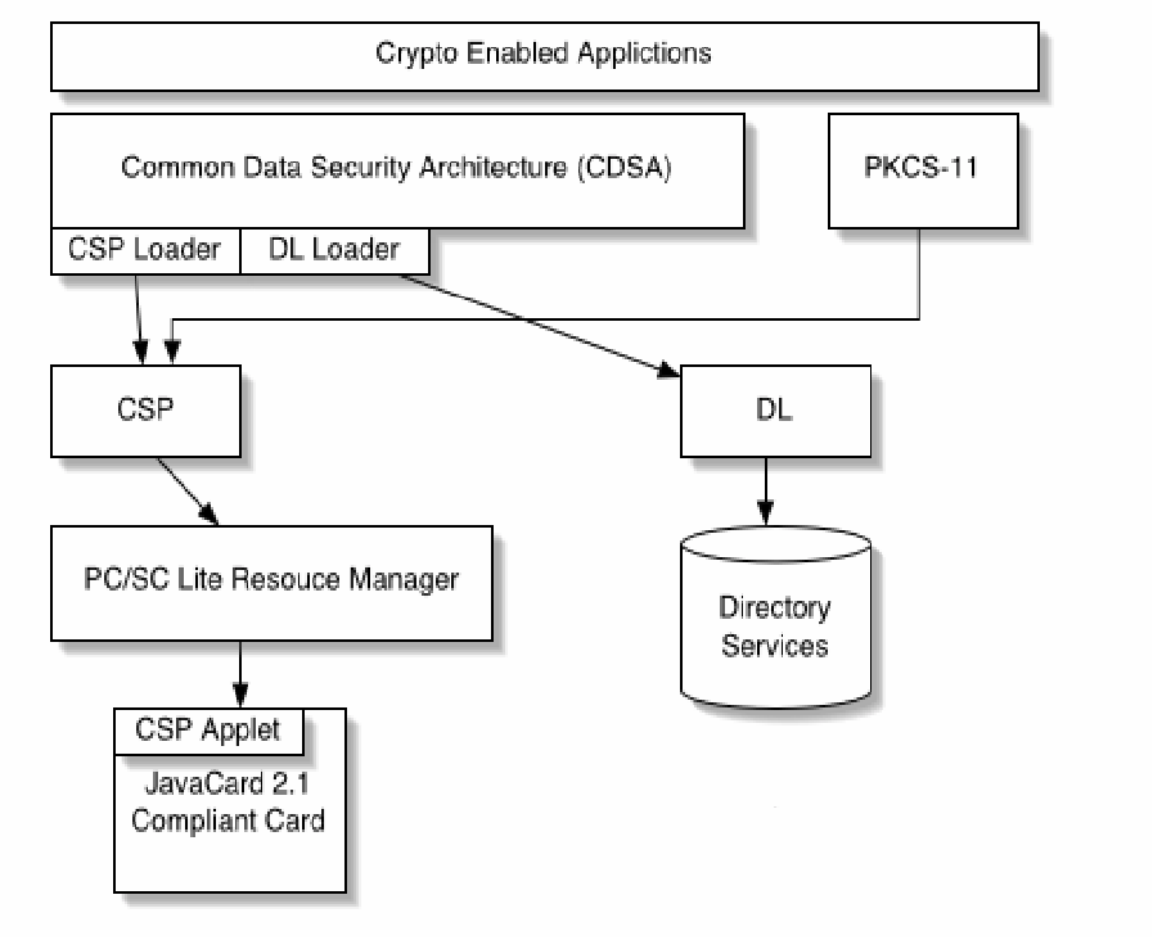


Figure 8.1. Muscle Architecture

Smart cards require large amounts of complex middleware that communicates with the card and exports the card's functionality to the host (Figure 8.1). These cards typically vary from release to release so this middleware generally is in constant change. Currently each card must have its own CSP (crypto/card service provider) on the host

creating large support problems and security trust well beyond most OS vendor's preferences.

Using a cryptography applet approach, it is required that only one host CSP be written for the middleware, thus reducing the time spent migrating to new card releases and vastly reducing the number of CSP's on the host.

8.3 Security Model

An identity number refers to one of 16 mechanisms (at maximum) by which the card can authenticate external applications running on the host. Each mechanism can be:

- Based on a PIN verification: identity numbers from 0 to 7 (PIN-identities) that are associated to PIN numbers from 0 to 7
- Based on a challenge/response cryptographic protocol: identity numbers from
- 8 to 13 (strong identities) that are associated to key numbers from 0 to 5
- Reserved for alternative authentication2 schemes: identity numbers 14 and 15

After an authentication mechanism has been run successfully, the corresponding identity is said to be “logged in”. Each identity is associated a counter for the maximum number of times an authentication mechanism can be run unsuccessfully for that identity. On a successful authentication the counter is reset. On an unsuccessful authentication the counter is decreased and, if it goes to zero, the corresponding identity is blocked and can not be logged in anymore. PIN codes have an unblock mechanism.

A PIN-identity login requires a PIN code verification. The PIN number is the same as the identity number. Strong identities involve use of cryptographic keys. Strong

identity n.8 requires use of key n.0, identity n.9 requires key n.1, and so on up to identity n.13. Login mechanisms for identities 14 and 15 are not specified in this release of the Card Edge specifications.

Each key or object on the card is associated with an Access Control List (ACL) that establishes which identities are required to be logged in to perform certain operations. The security model is designed in such a way to allow at least four levels of protection for card services:

- *No protection:* the operation is always allowed; in such a case the ACL requires only the anonymous identity to be logged in for the operation
- *PIN protection:* the operation is allowed after a PIN verification; in such a case the ACL requires a PIN-based identity to be logged in for the operation
- *Strong protection:* the operation is allowed only after a cryptography based, strong authentication of the host application (and optionally a PIN based authentication of the user); in such a case the ACL requires a strong identity to be logged in for the operation (and optionally a PIN based one)
- *Full protection* (operation disabled): the operation is never allowed.

The use of a private key on the smartcard is usually PIN protected, but some applications could require a strong protection. Reading of a private key is usually disabled. Public objects may be always readable, but their modification could be PIN protected. Private objects could require PIN protection for reading and protection with another PIN or strong protection for writing.

CHAPTER 9

SCAF SERVICE SECURITY FRAMEWORK

9.1 Introduction

Smart Card Authentication and Authorization Framework is a Smart Card based framework for SORCER that provides user authentication and authorization. This standard security mechanism enforces more consistent security policies and application developers are freed from the low-level drudgery of building explicit security controls into their software. To use the framework the only thing that a developer needs to do is extend the façade class to this framework. All the background communications and security is taken care of by the framework and a developer can build any application on top of SCAF.

Authentication is provided by Smart Card. A user has to present the password for the card and is taken to the main application only if that user pin can be verified from the card. The communication between the provider and the requestor is encrypted using SSL protocol so that the communication channel is completely safe from forgery and leakage. All the data that is sent from the requestor to the provider is signed using the user credentials on the card, so that any transaction can not be repudiated at a later date. This signed data is stored in a database and can be used to verify any repudiation claims. Authorization is provided by the grants defined in the policy files.

9.2 Objective/Approach

A grid is a vast repository of services. A security framework is very important for any grid since the requestors and services are at high risk. User credentials saved on servers or in files are not completely safe and are not portable. They are not intrusion protected and are not portable. Mutual authentication does not scale with large number of requestors and Authorization does not scale with large number of services. Smart cards are needed to provide reliable credentials for mutual authentication and authorization in Service-Oriented Environment.

The objective of SCAF is to

- Select CA to support PKI infrastructure
- Define use cases for SCAF
- Define user credentials for SO Computing
- Architect SO environment for SCAF
- Design the framework for authentication, authorization, and non-repudiation
- Design a service requestor and provider to validate SCAF

And the approach taken to achieve the objective was to

- Deploy CA to issue provider/requestor credentials
- Populate requestor credentials on smart card
- Develop and deploy SCAF in the SORCER environment
- Develop a service-oriented application: Bulletin Board
- Validate the framework with the Bulletin Board application

9.3 CardEdge Applet

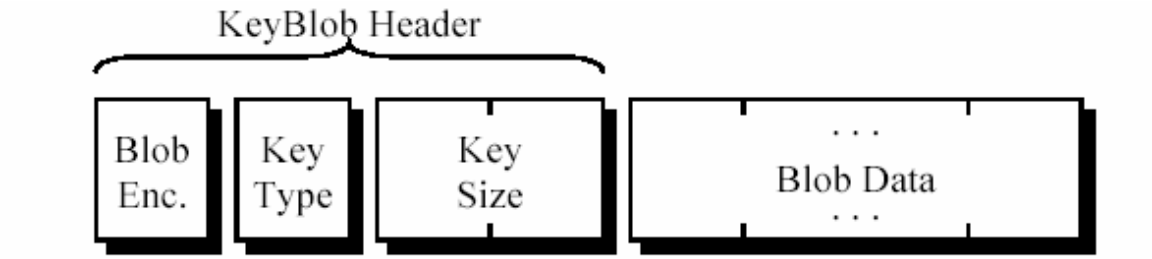
CardEdge Applet is the smart card application that acts as an interface to access the user credentials saved on the card. Once this applet is selected all the APDUs sent to the card are handed over to it by the operating system. This applet is responsible for providing access to keys and certificates stored on the system. Keys can be used for encryption and decryption. Before any information can be accessed from the card, a user has to verify its identity to the card, by presenting a PIN. After the PIN has been verified the user is granted access to perform operations (which ever are allowed). For reading keys (Public keys only) and certificates they need to be first exported as objects. The exported objects remain in RAM on card and can be read by the application on host system. Appendix A describes command APDUs to be exchanged between the card and the host computer. For each command, parameters that are to be provided as input and their format and what parameters are to be expected as output and their format are specified. Appendix B shows all the possible status words returned from the Applet commands, along with a symbolic name and a short description.

9.3.1 Key Blobs

A key blob is a sequence of bytes encoding a cryptographic key or key pair for import/export purposes. Whenever a key or key pair is transferred to the card, the application first transfers the corresponding key blob into the input temporary object then invokes the ImportKey command referencing it. Conversely, on a key or key pair export

operation, the application first invokes an ExportKey operation, and then retrieves the key blob from the output temporary object. Format of the key blob is shown in Figure 9.1

```
KeyBlob:
          Byte          Blob Encoding;
          Byte          Key Type;
          Short         Key Size; // In bits
          Byte[]        Blob Data;
```



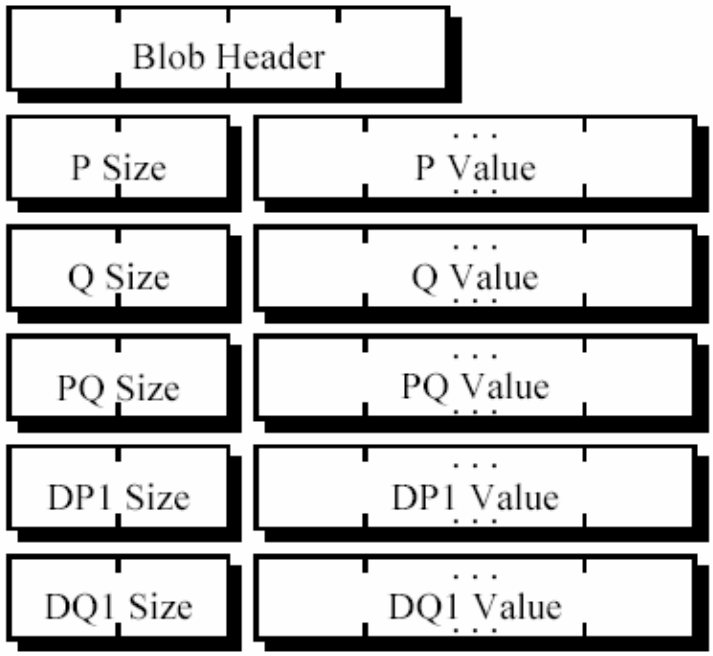

```
Values for Blob Encoding:
          0x00          BLOB_ENC_PLAIN;
          0x01          BLOB_ENC_ENCRYPTED (RFU)

Values for Key Type:
         RSA_PUBLIC          0x01     Public RSA key
         RSA_PRIVATE         0x02     Private RSA key
         RSA_PRIVATE_CRT     0x03     Private RSA CRT key

Allowed Values for Key Size:
         RSA             512, 768, 1024, 2048 …
```

Figure 9.1. Key Blob Format

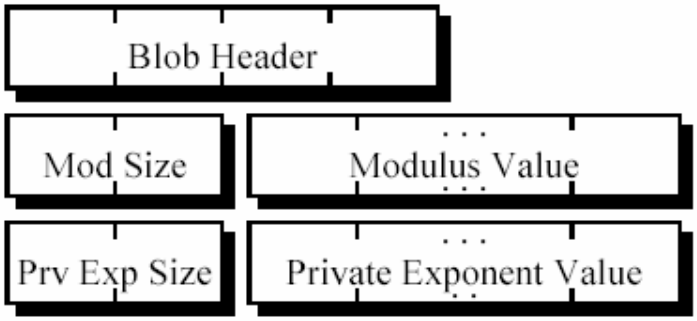
- RSA Key Blob

We only use RSA keys for the framework and so information related to only RSA keys has be explained in this document. Information for DSA and DES keys has not been mentioned here. Figure 9.2 (next page) shows RSA key blob definitions.

Key Type RSA_PRIVATE_CRT



Key Type RSA_PRIVATE



Key Type RSA_PUBLIC

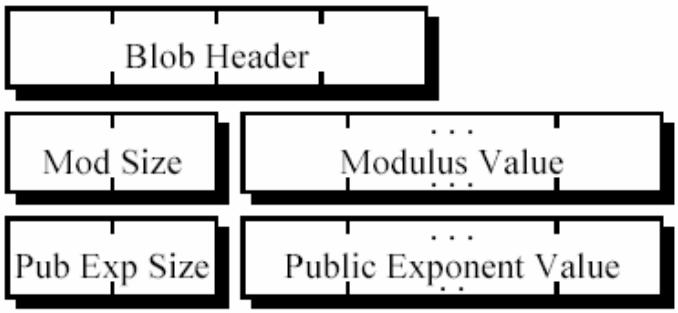


Figure 9.2. RSA Key Blob Definition

9.3.2 Application Directory Contents

PKCS #15 defines the standard for saving credentials on a token. Token in our case is a Java Card. PCKS #15 file format specifies how certain abstract, higher level elements such as keys and certificates are to be represented in terms of more lower level elements such as IC card files and directory structures. The format also suggests how and under which circumstances these higher level objects can be accessed by external sources and how these access rules are to be implemented in the underlying representation (i.e. the card's operating system).

Contents of Directory file for PKCS #15 are shown in Figure 9.3 and are explained below.

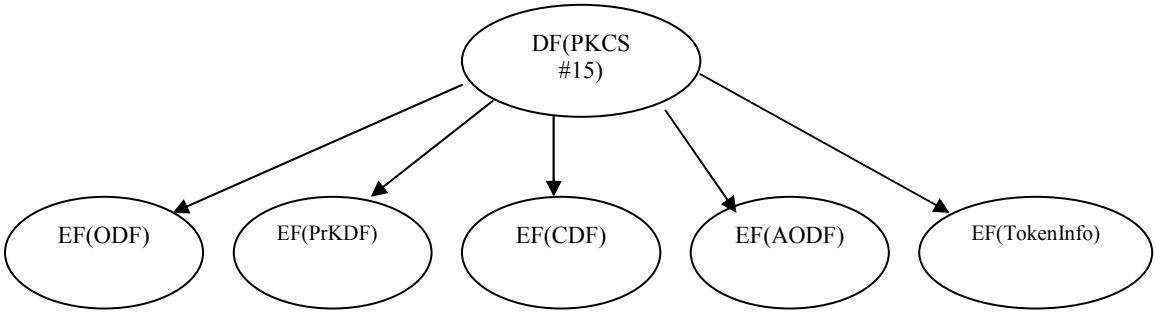


Figure 9.3. Contents of DF(PKCS #15)

- EF(ODF)

The mandatory Object Directory File (ODF) is an elementary file, which contains pointers to other EFs (PrKDFs, PuKDFs, SKDFs, CDFs, DODFs and AODFs), each one containing a directory over PKCS #15 objects of a particular class.

- Private Key Directory Files (PrKDFs)

These elementary files can be regarded as directories of private keys known to the PKCS #15 applications. They are optional, but at least one PrKDF must be present on an IC card which contains private keys (or references to private keys) known to the PKCS #15 application. They contain general key attributes such as labels, intended usage, identifiers, etc. When applicable, they also contain cross-reference pointers to authentication objects used to protect access to the keys. Furthermore, they contain pointers to the keys themselves. There can be any number of PrKDFs in a PKCS #15 DF, but it is anticipated that in the normal case there will be at most one. The keys themselves may reside anywhere on the card.

- Public Key Directory Files (PuKDFs)

These elementary files can be regarded as directories of public keys known to the PKCS #15 applications. They are optional, but at least one PuKDF must be present on an IC card which contains public keys (or references to public keys) known to the PKCS #15 application. They contain general key attributes such as labels, intended usage, identifiers, etc. Furthermore, they contain pointers to the keys themselves. When the private key corresponding to a public key also resides on the card, the keys must share the same identifier. There can be any number of PuKDFs in a PKCS #15 DF, but it is anticipated that in the normal case there will be at most one. The keys themselves may reside anywhere on the card.

When a certificate object on the card contains the public key, the public key object and the certificate object shall share the same identifier. This means that in some cases three objects (a private key, a public key and a certificate) will share the same identifier.

- Secret Key Directory Files (SKDFs)

These elementary files can be regarded as directories of secret keys known to the PKCS #15 applications. They are optional, but at least one SKDF must be present on an IC card which contains secret keys (or references to secret keys) known to the PKCS #15 application. They contain general key attributes such as labels, intended usage, identifiers, etc. When applicable, they also contain cross-reference pointers to authentication objects used to protect access to the keys. Furthermore, they contain pointers to the keys themselves. There can be any number of SKDFs in a PKCS #15 DF, but it is anticipated that in the normal case there will be at most one. The keys themselves may reside anywhere on the card.

- Certificate Directory Files (CDFs)

These elementary files can be regarded as directories of certificates known to the PKCS #15 applications. They are optional, but at least one CDF must be present on an IC card which contains certificates (or references to certificates) known to the PKCS #15 application. They contain general certificate attributes such as labels, identifiers, etc. When a certificate contains a public key whose private key also resides on the card, the certificate and the private key must share the same identifier. Furthermore, certificate directory files contain pointers to the certificates themselves. The certificates themselves may reside anywhere on the card.

- Data Object Directory Files (DODFs)

These files can be regarded as directories of data objects (other than keys or certificates) known to the PKCS #15 applications. They are optional, but at least one DODF must be present on an IC card which contains such data objects (or references to such data objects) known to the PKCS #15 application. They contain general data object attributes such as identifiers of the application to which the data object belongs, whether it is a private or public object, etc. Furthermore, they contain pointers to the data objects themselves. The data objects themselves may reside anywhere on the card.

- Authentication Object Directory Files (AODFs)

These elementary files can be regarded as directories of authentication objects (e.g. PINs, passwords, biometric data) known to the PKCS #15 application. They are optional, but at least one AODF must be present on an IC card, which contains authentication objects restricting access to PKCS #15 objects. They contain generic authentication object attributes such as (in the case of PINs) allowed characters, PIN length, PIN padding character, etc. Furthermore, they contain pointers to the authentication objects themselves (e.g. in the case of PINs, pointers to the DF in which the PIN file resides). Authentication objects are used to control access to other objects such as keys. Information about which authentication object that protects a particular key is stored in the key's directory file, e.g. PrKDF.

- EF(TokenInfo)

The mandatory TokenInfo elementary file with transparent structure shall contain generic information about the card as such and it's capabilities, as seen by the PKCS15

application. This information includes the card serial number, supported file types, algorithms implemented on the card, etc.

- EF(UnusedSpace)

The optional UnusedSpace elementary file with transparent structure is used to keep track of unused space in already created elementary files. When present, it must initially contain at least one record pointing to an empty space in a file that is possible to update by the cardholder.

- Other elementary files in the PKCS #15 directory

These (optional) files will contain the actual values of objects (such as private keys, public keys, secret keys, certificates and application specific data) referenced from within PrKDFs, SKDFs, PuKDFs, CDFs or DODFs.

9.4 Signing and Verification

9.4.1 Signing Process

The whole signing process [21] can be broadly divided in to following steps.

1. Initialize the token
2. Compute hash (digest) of the data to be signed
3. Sign the hash using private key
4. Retrieve the certificate corresponding to the private key used for signing
5. Clean-up

- Initialize the token:

To start the signing process we need to initialize the token. This is done in order to determine the number of cards and key-pairs present on the token. As a secret PIN protects the access to private key on the token, the PIN needs to be supplied to the token. This process also starts a new session on the token.

- Compute hash:

After initializing process we need to calculate the digest (hash) of the data to be signed.

- Sign the Hash:

This step is responsible for generating the signature for the hash computed by the previous step. As the private key never leaves the PKCS token, the data to be signed (hash in our case) needs to be passed on to the PKCS token.

- Retrieve Certificate:

Once we have the signature, we need to retrieve the corresponding certificate, so that it could be sent along with signature for verification.

- Clean-up:

Finally we need to close the current session to free up any resources being used by the system.

9.4.2 Verification Process

Verification of the signature [21] using public key through PKCS#11 compliant token is carried out in following steps.

1. Initialize the token
2. Compute hash (digest) of the data to be verified
3. Verify the hash using public key
4. Validate the certificate chain.
5. Clean-up

- Initialize the token:

First we need to initialize the token for verification. Here we pass the public key, which is used for verifying the signature.

- Compute Hash:

As during signing process only the hash of data was signed, if we try to verify the document content instead of its hash, the verification process would fail. Hence we need to compute hash in the same manner we did for signing.

- Verify the signature:

Once we have the hash we can validate the data using the public key supplied during initialization step. This step returns true if the verification was successful else it returns false.

- Validating a Certificate

Verification step only verifies the integrity of the document, it only tells us that the document was signed by a particular key-pair and it has not been modified after signing. It is always advisable to first verify the certificate before verifying the data itself. This helps us in determining whether the certificate is authentic or was tampered with.

Important thing to remember about verification step is that it does not validate the identity of the signer. Authenticating the signer information is critical to prevent anyone generating key-pairs with some one else's identity. This is ensured by a Certificate Authority (CA) who carries out reasonable identity checks before issuing a certificate to a person or company. The CA will have its own key pair and its own certificate, which is self signed, also known as a root certificate. The CA's private key is used to sign all issued certificates. Sometimes a CA will be a 'channel CA' to some other certifying authority and thus a subsidiary. In such a case the signatures can be traced up along certificate chain to a root certificate authority. The root certificate is self-signed. Unless the entire certificate chain is validated, a given certificate and a given signature cannot be guaranteed to be valid.

To validate a certificate, one needs to validate the digital signature on it. This requires parsing the X.509 certificate and identifying the content and the digital signature. Once the data content and the digital signature are located, the data must be hashed to obtain a digest. Next, the certificate authority's root CA certificate must be parsed to extract the public key. This certificate is typically placed in a trusted root certificates repository. The public key is then used to decrypt the digital signature on the certificate and reveal the hash. This hash must match with the computed hash on the data.

- Clean-up:

Finally we need to close the current session to free up any resources being used by the system.

9.5 Channel Confidentiality In SCAF

SSL authentication [20] assures authentication on both ends. It not only encrypts the data but determines whether or not each party (server and client) has the expected authentication. Though secure certificates can be created independently, getting a secure certificate from a validated certificate authority helps to ensure the parties are trusted. The details of SSL communication are hidden to the system developer and the support is provided to us by the underlying Jini technology.

SSL communication requires both ends to contain matching private key and public which are required for encryption and decryption over the secure channel. As the user credentials in SCAF are stored on a smart card and there is no way the private can be read or transported so we use a proxy certificate. This certificate initially is an unsigned and is transferred in the jar file that is downloaded dynamically on the requestor side. This certificate is then signed using the private key on the card. Private key and public key of this signed certificate are used as the private credential and public credential, respectively, for the subject that is used to make SSL tunnel. After the SSL tunnel has been established the signed certificate here is not used. Data that is sent from the requestor to the provider is signed using the key on the Smart Card. The principal that is used for authorization on provider contains the common name from the certificate read from the card. This principal is sent in the context with the exertion.

9.6 Benefits

- Safeguarding user integrity, passwords are safely stored on smart cards

- Preventing credential intrusion
- Scalability for unlimited number of providers/requestors
- Cost effective with own CA
- Reliable authentication in SO frameworks
- Flexible multilayered authorization using JAAS
- Secure communication between ServiceUI and Service Provider
- Trusted zero install requestor ServiceUI with JavaCardLoginModule

9.7 Framework Design

9.7.1 SCAF

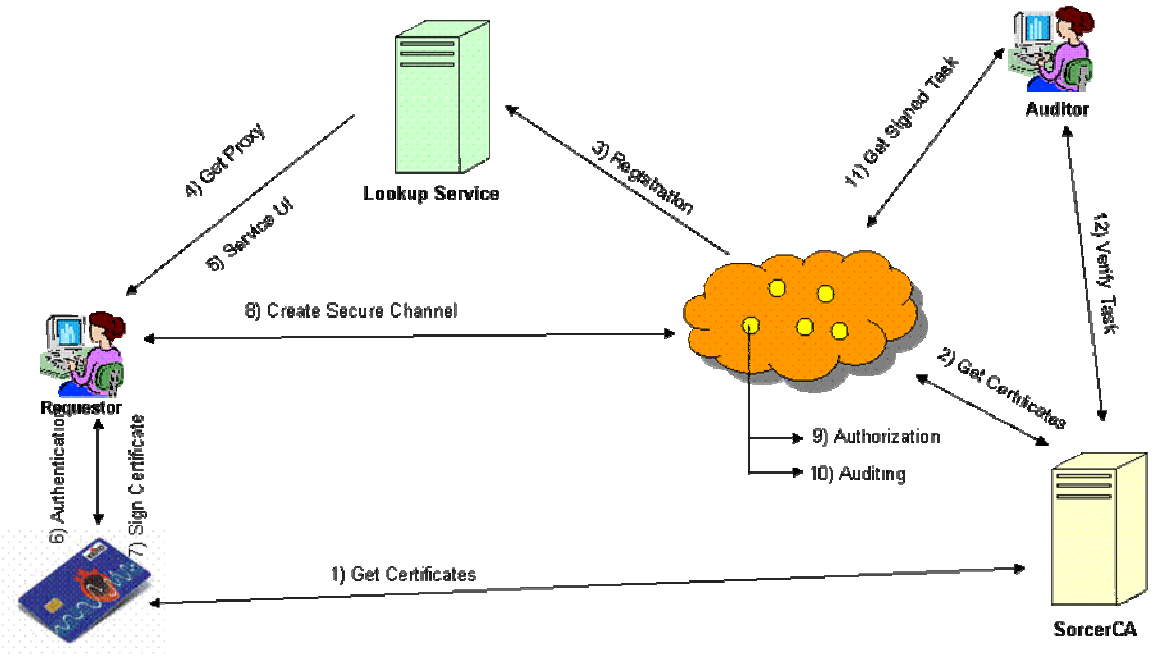


Figure 9.4. SCAF

General functioning of the framework is explained in Figure 9.4. SCAF is explained below without referring to Java classes used for framework implementation. Certificates are first downloaded from the root certificate authority (SorcerCA) on to the card. These credentials are used by the requestor for authentication, signing proxy certificate for creating SSL tunnel with provider and signing data for auditing. Provider also uses certificates issued by this root certificate authority but certificates in this case are stored in a keystore. Provider starts up using the credentials from keystore and registers with one or many lookup services. A requestor queries one or many lookup services to get the proxy for the provider it wants to communicate to. Service user interface is then shown to the requestor which takes the password for the card. This password is used for authentication. Once a user has been authenticated proxy certificate is signed using private key on card. Private key and public key of this proxy certificate are used as matching key pair for creation of SSL tunnel with the provider. Any task that needs to be executed by provider is sent as a signed service task so that it can be used for auditing. Every signed service task received by provider for execution is audited. Request for task execution from service user interface to provider is then authorized using permissions granted in the policy file. If proper permission are defined for the principal in the policy file then the task is executed and results are sent back to the requestor. Otherwise illegal access exception is thrown. If a repudiation claim is made at a later time about some transaction then database query tool provided with the framework can be used to verify whether or not the task was signed by the principal associated in database with signed service task.

9.7.2 Certification Process

Certification process starts when a user submits personal information like name, email address etc to Registration Authority (RA). PKI user is then registered with the RA. After registration, key pair is generated for the use, depending on the security device selected. Security device could be a software security device, if the certificate is required to be saved in the browser or it could be any other security module like one from MUSCLE (pkcs#11.dll) if the key pair is required to be generated on the card. The latter approach is used in this research and is shown in Figure 9.5.

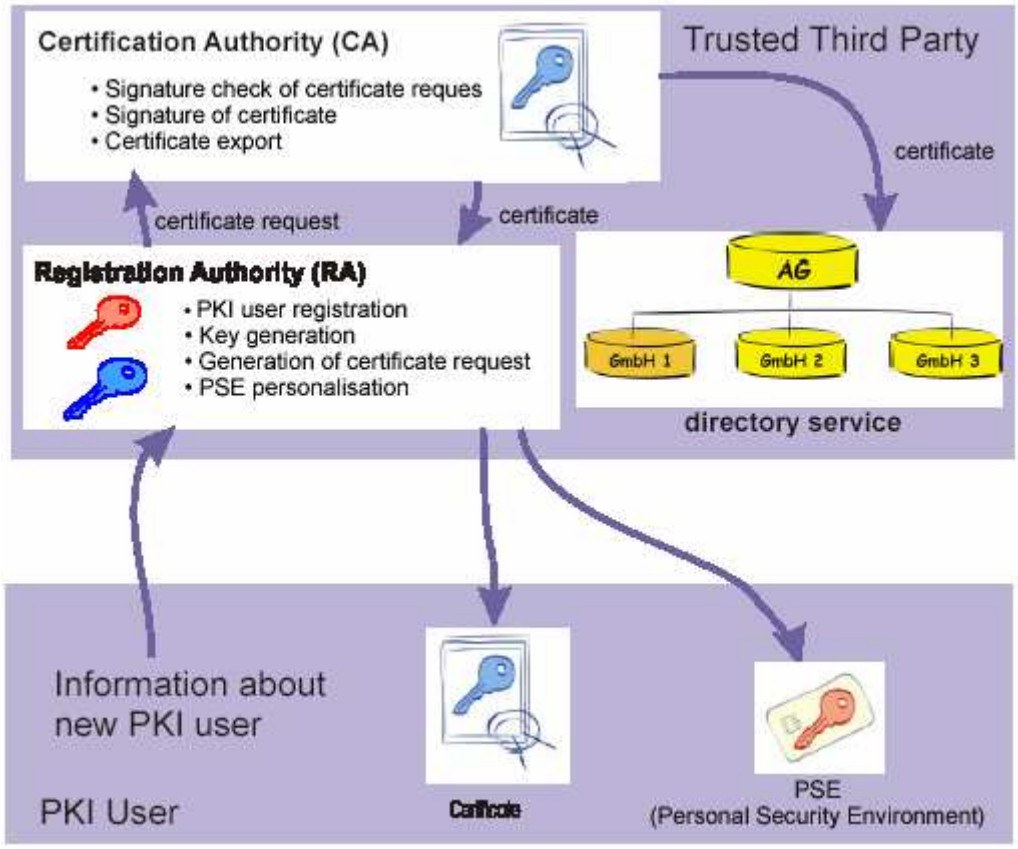


Figure 9.5. Certification process

After the key pair generation a certificate request is generated that is sent to the Certificate Authority (CA). At CA the signature for the request is checked to make sure that the request came from a valid RA. Signature for user certificate is created if a valid request is found and the certificate is stored in a database (MySQL) and is exported back to RA from where is can be picked up by the user and can be stored using the module defined in security devices.

9.7.3 Authentication

Authentication is accomplished using JAAS. Any application that uses SCAF and requires a service user interface will extend SecureUI, the façade class to SCAF. The application after calling the constructor will call the function to show the login frame, which requires a user to enter password for card. After the password has been submitted a LoginContext instance is created. Login Context class requires two parameters; configuration file name and CardCallbackHandler object. CardCallbackHandler is instantiated by passing it the password entered by user. The LoginContext consults a configuration that determines the authentication service, or LoginModule, that gets plugged in under that application. LoginModule in SCAF is CardLoginModule. LoginContext performs the authentication steps in two phases.

In first phase, or the 'login' phase, the LoginContext invokes the configured CardLoginModule and instructs to attempt the authentication only. Authentication here succeeds if the system is able to verify the password on card. CardLoginModule gets the password from CardCallbackHandler using CardCallback object. If CardLoginModule

successfully pass this phase, the LoginContext then enters the second phase and invokes the configured CardLoginModule again, instructing each to formally 'commit' the authentication process. During this phase CardLoginModule associates the relevant authenticated principals and credentials with the subject. As explained before the authenticated principals and credentials are those of a proxy certificate. This proxy certificate is signed by the user's private key on the card. If either the first phase or the second phase fails, the LoginContext invokes the configured CardLoginModule and instructs each to 'abort' the entire authentication attempt.

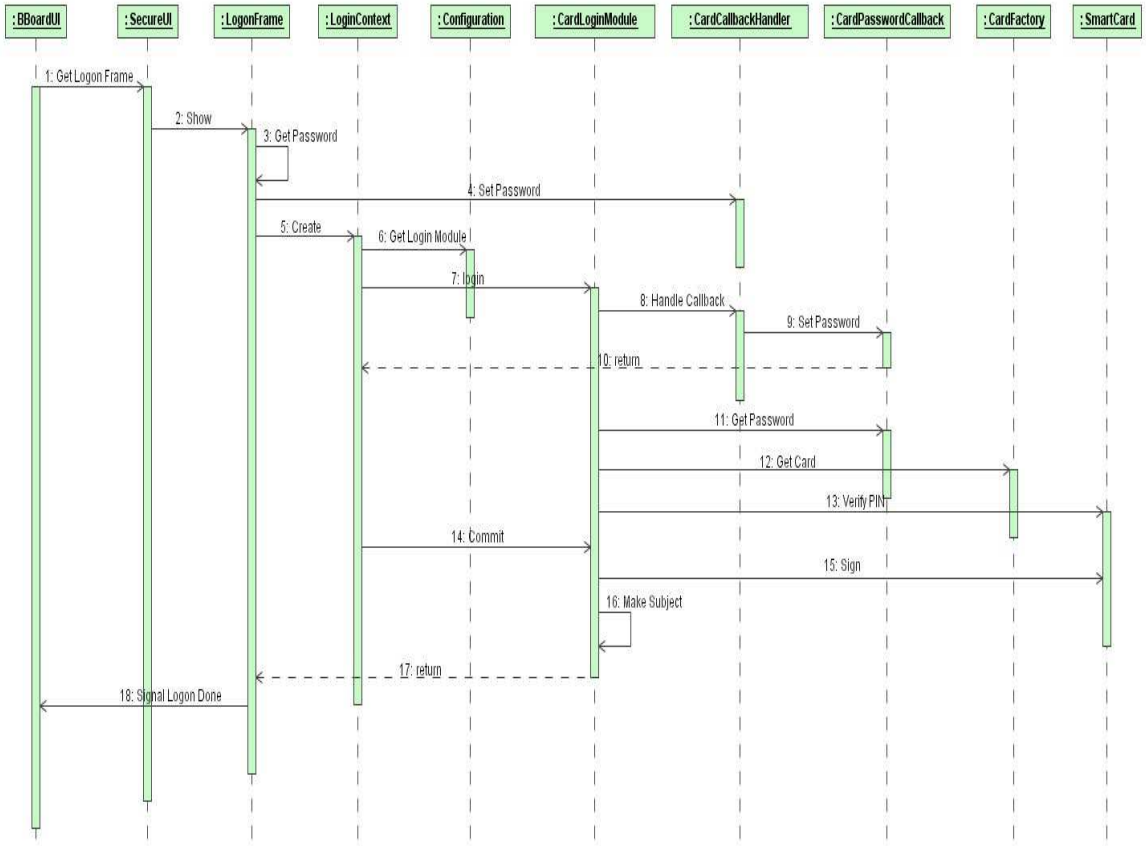


Figure 9.6. Authentication Sequence Diagram

CardLoginModule then cleans up any relevant state they had associated with the authentication attempt. Authentication sequence diagram [26] is shown in Figure 9.6. If user is able to authenticate and credentials can be added to the subject successfully, SecureUI signals logon done to the application passing password and subject just created. This subject contains two principal names, one for the proxy certificate and the other for the certificate on the card. Former principal is used to make SSL connection and the latter is used to authorize user on provider end. Each of this password and subject can be used by application at any time. Class diagram [26] for authentication is shown in Figure 9.7.

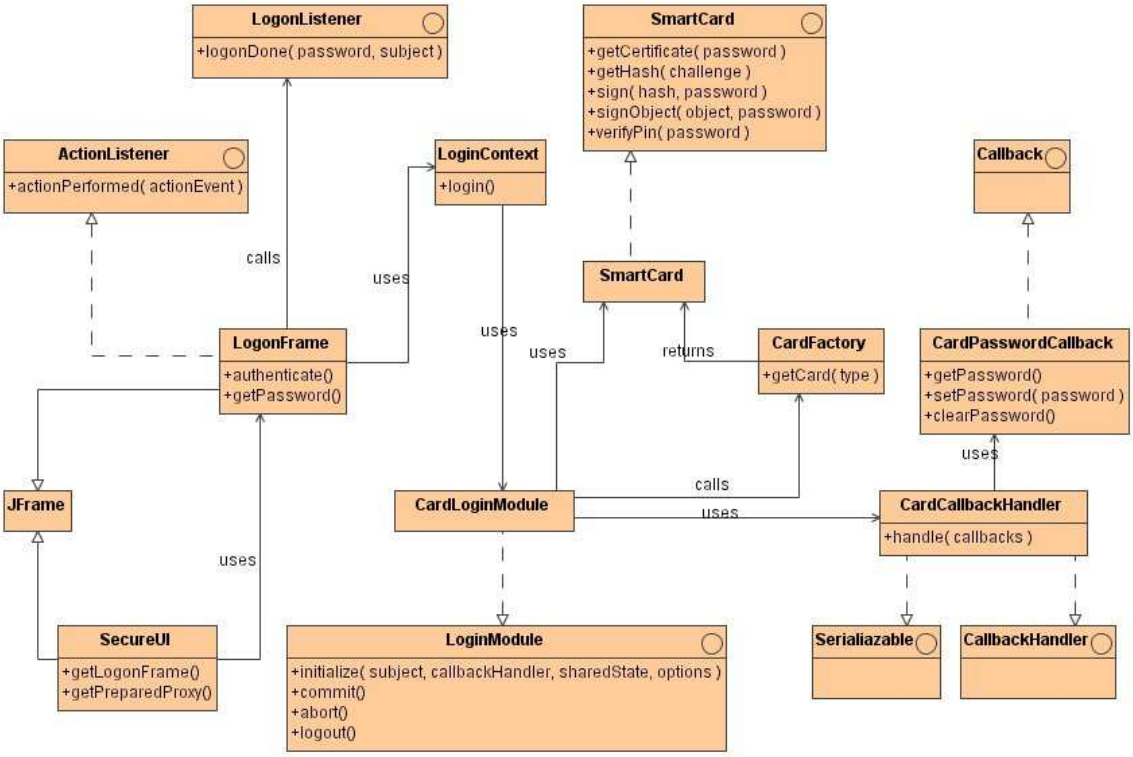


Figure 9.7. Authentication Class Diagram

9.7.4 Authorization

Once authentication has successfully completed, JAAS provides the ability to enforce access controls upon the principals associated with the authenticated subject. But this access control is not forced on the requestor side. After authentication, application takes the input from the user and makes a task out of it. This task is then executed by SORCER framework. The task created here is signed using private key on the card. This process is explained in next section. SCAF providers implement the access control model of security, which defines a set of protected resources, as well as the conditions under which named principals may access those resources. JAAS follows this model, and defines a security policy to specify what resources are accessible to authorized principals.

As explained above, principal from proxy certificate is only used to make SSL tunnel and principal from certificate on card is used for authorization. To accomplish this application needs to send the latter principal in the service task's context. SORCER framework receives this service task and retrieves the name for the method that needs to be executed. The SORCER framework then checks in task context for a principal that would be used for authentication. If there is no principal found then NoPrincipalException is thrown and task is not executed. If the required principal is found in the context then a new subject is created using that principal. SORCER framework then executes the method using the subject just created. JAAS framework then checks the permission for the principal from the policy file. When security checks occur during execution, the Java 2 SecurityManager queries the JAAS policy, updates the current AccessControlContext with the permissions granted to the subject and the

executing codesource, and then performs its regular permission checks. If the permission has not been defined for the principal in the subject then IllegalAccessException is thrown. Otherwise the method is executed and the results are sent back to requestor application.

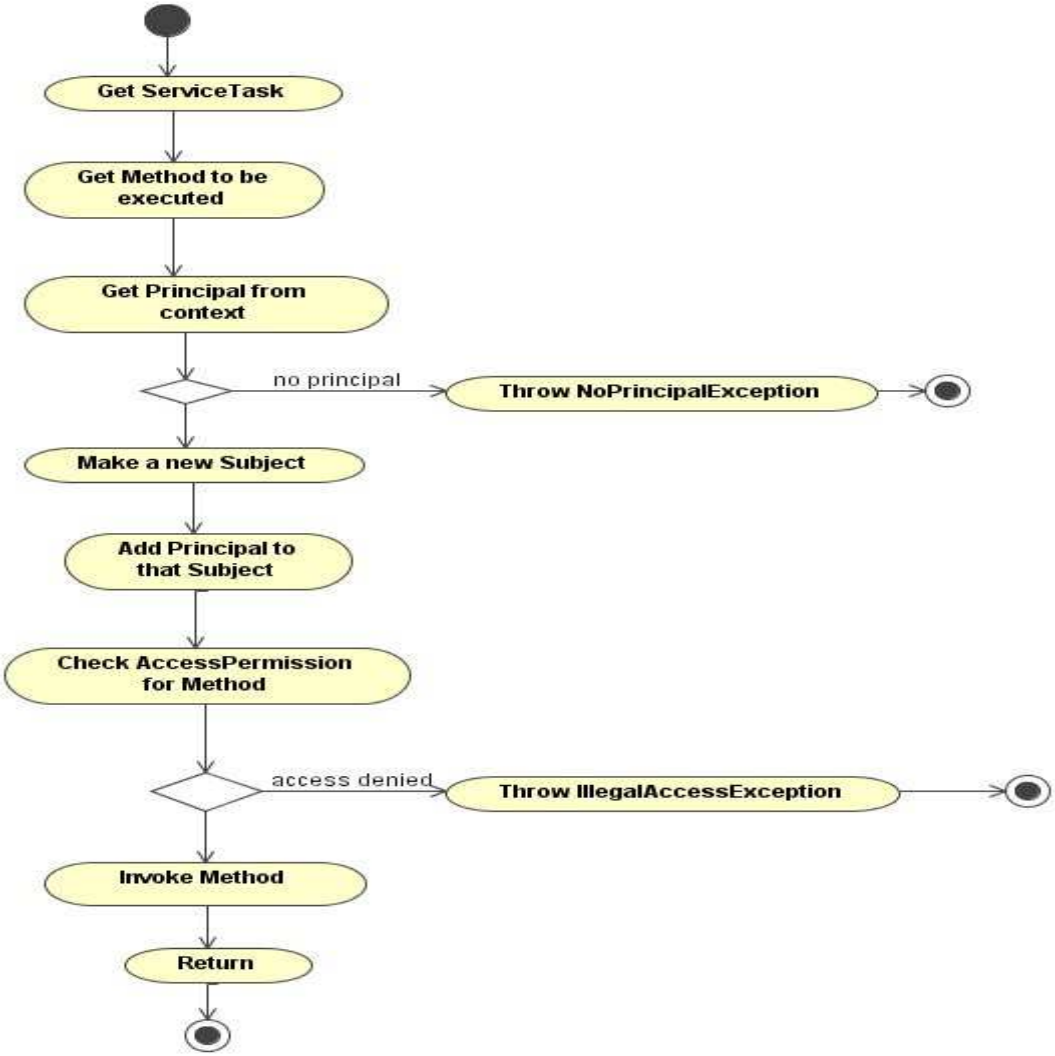


Figure 9.8. Authorization Activity Diagram

Activity diagram [26] for authorization is shown in Figure 9.8.

9.7.5 Non Repudiation

Non Repudiation means that the requestor shall sign the communication sent by it. Once an entity has signed the message, it cannot repudiate it. This is achieved using SignedServiceTask. Any application that uses SCAF and wants to provide non repudiation feature, uses SignedServiceTask instead of ServiceTask as exertion. ServiceTask is signed using the private key on the card and the signature and ServiceTask object are saved in SignedServiceTask. ServiceTask is saved as a MarshalledObject and the signature is saved as an array of bytes. SignedServiceTask implements SignedTaskInterface, which provides functions like getSignature, verify and getObject. Figure 9.9 shows a sequence diagram for auditing.

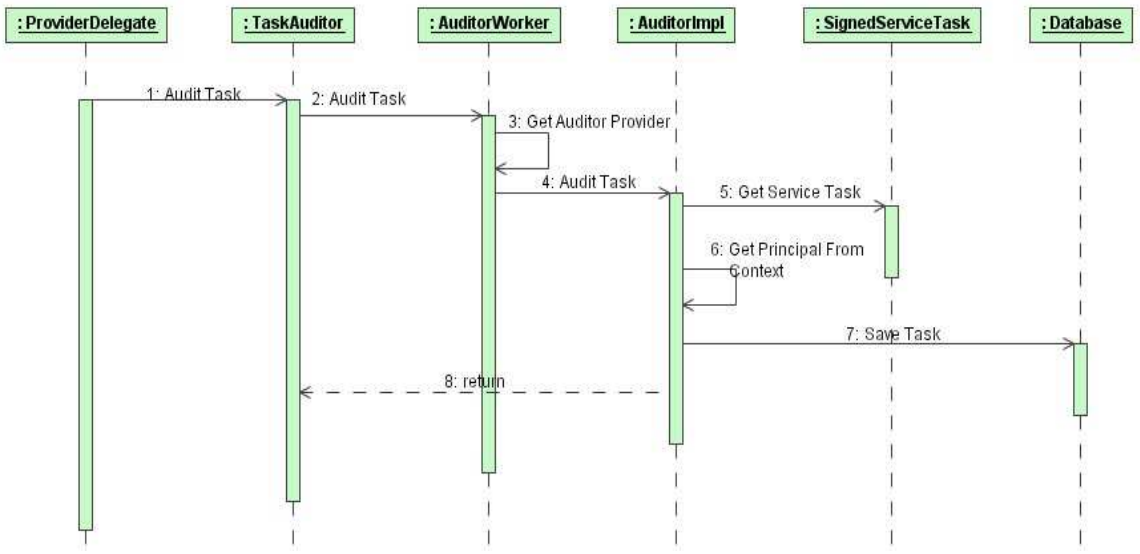


Figure 9.9. Auditing Sequence Diagrams

When SORCER framework (ProviderDelegate) receives SignedServiceTask it checks if it is an instance of SignedTaskInterface. If it is then the SORCER framework calls TaskAuditor to audit the task just received. ServiceContext is also sent to auditor

provider so that principal can be retrieved from it. After that the ServiceTask is retrieved from the SignedServiceTask and execution continues as it would in normal case.

TaskAuditor starts an AuditorWorker thread and gets auditor proxy. It then passes SignedServiceTask to the provider where the task is persisted as a sequence of bytes in a database along with the principal retrieved from the context.

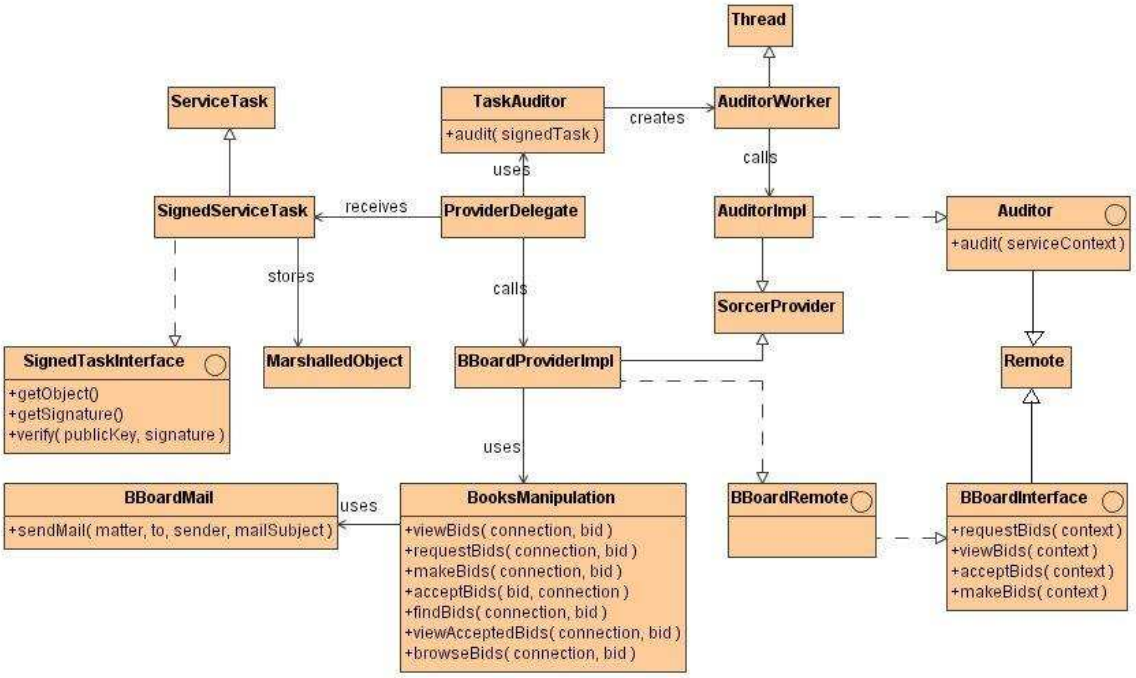


Figure 9.10. Authorization and Auditing Class Diagram

A JDBCQueryTool is provided with the framework which is used to verify repudiation claims. It shows all signed tasks saved in database along with date, time and principal who signed it. To check whether the principal that appears with the signed task is the one who submitted the task, user needs to select the row and then verify the task. Task is verified using the certificate for the principal that is saved in a truststore which contains certificate for all SCAF users. Public key from the certificate is used to verify if

the signature after decryption matches with the service task stored in signed service task. Authorization and Auditing class diagram is shown in Figure 9.10.

9.8 Implementation

9.8.1 Technical Architecture

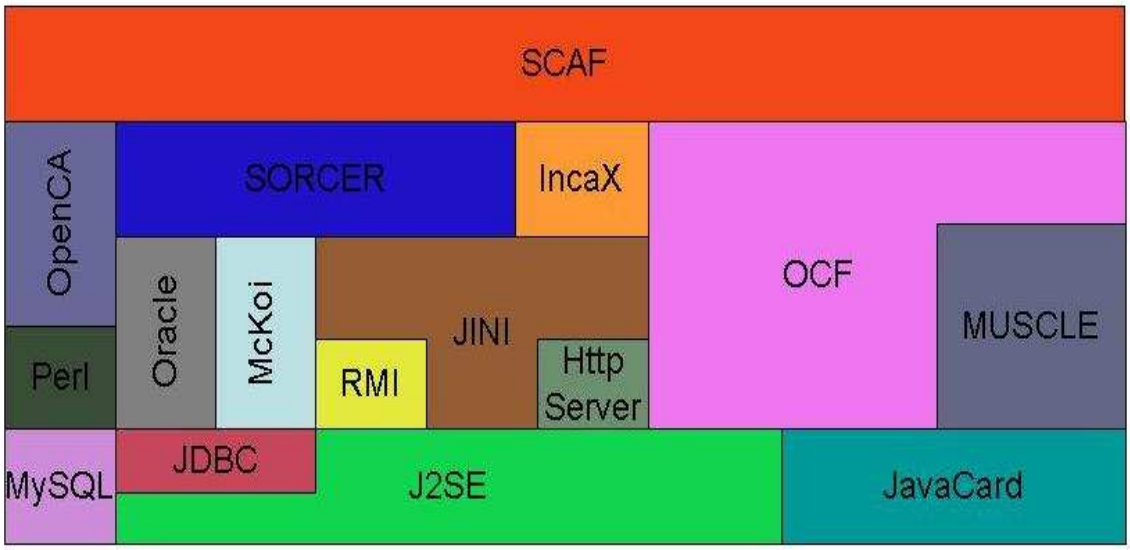


Figure 9.11. Technical Architecture

SCAF uses SORCER to provider service oriented framework. To display service user interfaces SCAF uses IncaX browser. SORCER and IncaX browser are both based on JINI network technology, which uses RMI for remote method calls and an Http Server (Tomcat) for downloading dynamic code. SORCER also uses two databases; Oracle and Mckoi [22]. Oracle is used by validation application and Mckoi is used as an embedded database from Auditor service. Services use JDBC to communicate with databases. Security is provided by J2SE. Smart Card used in SCAF is Javacard and has MUSCLE

applet installed on it. SCAF uses OCF to communicate with the applet installed on Javacard. Javacard used for SCAF is Schlumberger's Cyberflex Access 32K. Figure 9.11 shows the technical architecture for SCAF.

9.8.2 Package Diagram

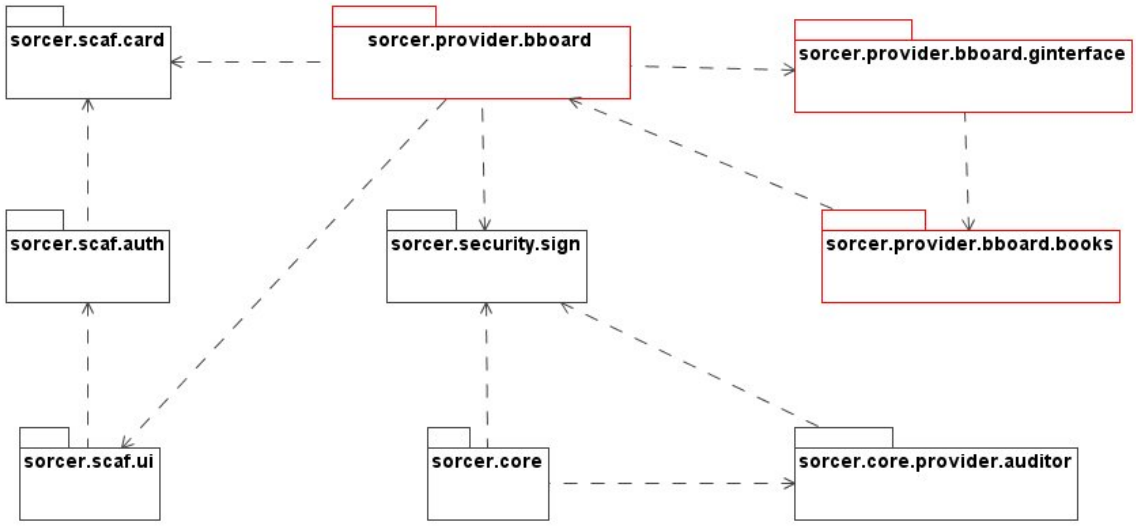


Figure 9.12. Package Diagram

SACF provides classes for authentication, for working with smart card and classes that are required to make the user interface. These classes are stored in packages sorcer.scaf.auth, sorcer.scaf.card and sorcer.scaf.ui respectively. The package sorcer.scaf.auth contains classes that are required by JAAS like CardLoginModule, CardPasswordCallback, CardCallbackHandler etc. Classes that are required to work with smart card, like SmartCard factory, JavaCard etc are in sorcer.scaf.card. SCAF also provides an auditing service which is in package sorcer.core.provider.auditor. Authorization is handled in ProviderDelegate which is in sorcer.core. Task that gets executed as excertion is required by SCAF to be submitted as SignedServiceTask which

is in package sorcer.security.sign. This package contains all the classes that are required to sign a task. Validation application, Bulletin Board is in package sorcer.provider.bboard. Complete package diagram [26] is shown in Figure 9.12.

9.8.3 Physical Architecture

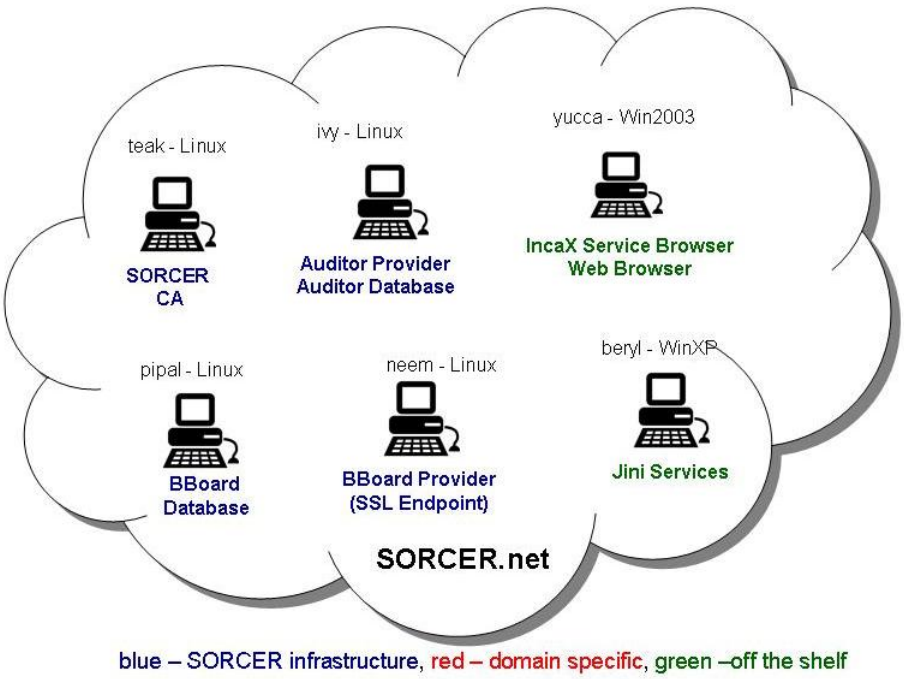


Figure 9.13. Deployment Diagram

The Jini architecture specifies a way for clients and services to find each other on the network and to work together to get a task accomplished. Jini technology provides a flexible infrastructure for interactions between clients and services regardless of their hardware or software implementations. And hence SORCER that is based on Jini technology is also distributed across many computers.

Jini core services like Lookup service, Javaspace service, Transaction service etc run on a Windows Xp computer. Auditor provider runs on a Linux machine. McKoi database that is required by Auditor provider runs on the same machine. Bulletin Board that uses SCAF runs on a separate Linux machine. Oracle database that is used by this provider runs on a separate Linux server. Finally IncaX browser, which acts as a user agent, runs on another Win2003 machine. Deployment diagram for SCAF which is based on SORCER is shown in Figure 9.13.

9.9 Validation

9.9.1 SCAF Credentials

To get key pair an installed/compiled OpenCA installation needs to be configured. There are three interfaces to an OpenCA installation; ca-node, ra-node, pub node for CA, RA and user repectively.

First of all Certification Authority needs to be initialized from ca-node. This is done only once and is skipped if CA has already been initialized First step in that is to initialize database. After that a new secret key is generated: des3 rsa 1024. This key is used to generate a new cert request. For SORCER a self signed CA Certificate is generated from already generated request. It could be signed by any other CA.

To generate a certificate for CA operator a new request is submitted with fields as appropriate. This request, as any other certificate request can be edited. After editing the request, certificate is issued which can be exported as PCKAS#12 to be saved on disk

and then imported into the browser. RA operator certificate can be issued likewise, only by changing the role to RA operator instead of CA operator.

RA gets initialized from ra-node. For that RA database needs to be initialized first. After that the configuration is imported from ca-node. To move the certificates down to RA from CA, from ca-node enroll all data to a lower level of the hierarchy. Then from ra-node download all data from a higher level of the hierarchy.

User certificate is requested from pub-node. User fills in all the required fields and submits request. A key pair is generated after the submission. For this thesis, pkcs#11 module provided by MUSCLE is used as a security device which generated private-public key pair on the card. Certificate request needs to be approved by RA to get the certificate from CA. RA exports the request to CA after approving the request by uploading request data to a higher level of the hierarchy. All requests are imported into CA by receiving request data from a lower level of the hierarchy. CA after performing all the validations approves and signs the certificate request. Certificate data thus generated is enrolled to a lower level of the hierarchy which is imported into RA by downloading certificate data from a higher level of the hierarchy. Certificate can be picked up from public interface by presenting either the certificate request number or the certificate serial number generated after it gets approved. If MUSCLE pkcs#11 module is loaded as a security device then the certificate would be stored on the card and can be used from there. Key pair and certificate saved on the card are used for authentication, authorization and non-repudiation explained in the following sections.

9.9.2 Bulletin Board Services

SCAF is validated using Bulletin Board application. This application aims to provide an online market place, where a user can request the bids for the product he/she wants to sell, make the bids for the product he/she is interested in buying and accept the bids, once a bid has been made for the request posted. Requests for books, computers, cars and roommates can be posted on the Bulletin Board.

Seller posts the request and waits for bids to be made for the requests and gets ID# for that. He can then view the bids made for the ID# given. Buyer can browse through the requests or he can find the requests matching his preferences. Buyer makes bid for the request. The request can be greater than or equal to the Min price specified by the seller. Once a bid has been made by a buyer, it starts showing up on the sellers interface. Seller can wait for other bids or he can accept the bids if the bid offer seems reasonable to him. Once a bid has been accepted it would start showing up on the buyers interface. A mail will be sent to the Seller, Buyer and the Administrator which contains the details of the bid. As explained in the previous section, all the tasks that are sent from requestor to provider in SCAF are signed using the private key on the card.

A user can be a seller or a buyer. A seller can request bids for the product he wants to sell, if the request is posted properly then the seller will get the ID# for the request. A seller can also view the bids that have been made for the ID#. A seller will finally accept the bids once the list of bids has been displayed in the interface. A buyer can find the request that have been posted on the Bulletin Board, browse through the

requests, make a bid for a particular request by specifying the ID# and view all his bids that have been accepted.

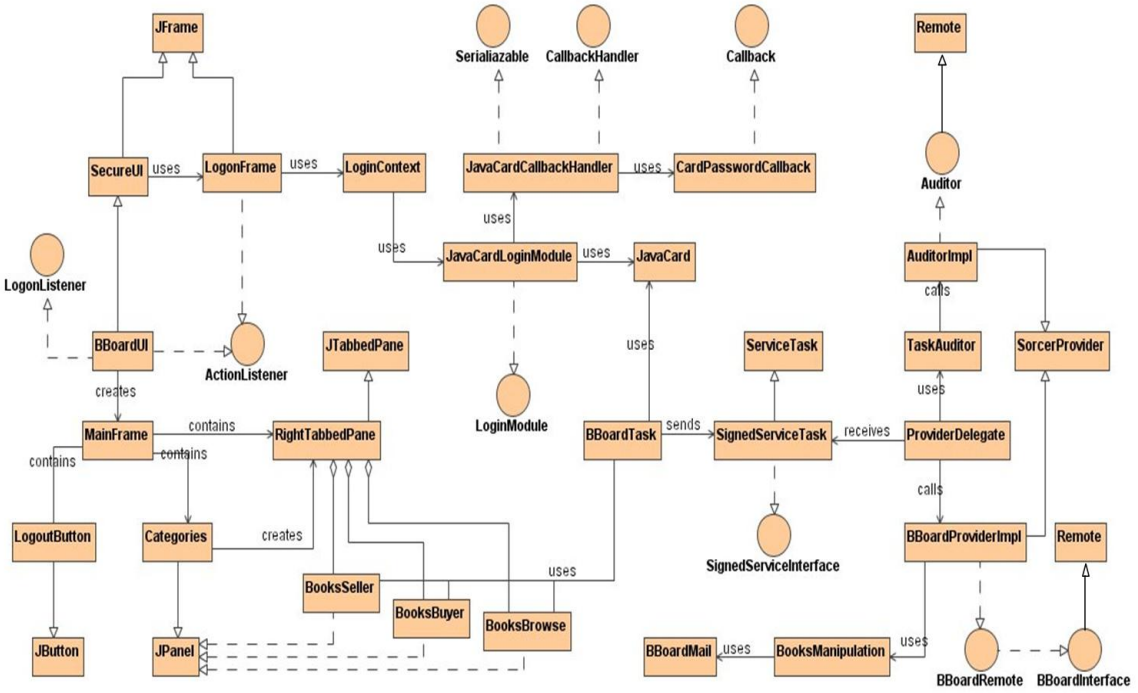


Figure 9.14. Bulletin Board Class Diagram

Class diagram for complete application is shown in Figure 9.14. It is a combination of class diagram for authentication, authorization, signing and auditing classes which comprise the whole system. Only one feature (Books) of Bulletin Board has been shown. Classes for other features like computers, cars etc follow the same naming convention.

9.9.3 Bulletin Board User Agent

User agent in Bulletin Board is a graphical user interface or ServiceUI which is sent from the provider along with proxy. ServiceUI can be viewed using any service browser.

IncaX service browser was used for this research. Client does not need to know where the provider is running or what are the classes required by the interface. All the classes are downloaded from the provider dynamically. Client only needs to have a ServiceUI browser installed. The provider can be started on machine on the network without even telling the client about the change. Only thing that is required on the client is the framework to support card operations. Open Card Framework classes need to be in the classpath of the client we are trying to run SCAF from. A truststore containing SORCER certificate authority's certificate is required be supplied to IncaX browser while starting it.

All interactions with provider are accomplished using this user interface. User interface can be logically divided into two parts; right panel and left panel. They can be referred to as work areas.

Left panel holds a tree which has main categories, like books, cars etc as nodes. These nodes act as a sub tree for operations provided for a particular category and contain those operations as sub nodes. Right panel is the main working area of the interface. It is a tabbed pane with tabs for each operation provided. User can browse through the interface using either the tree in left panel or the tabs in right panel. A seller user can either open bids for a new book or he can check and accept bids placed for any previously opened bids. Work area for seller provides interface for executing any of stated functionality.

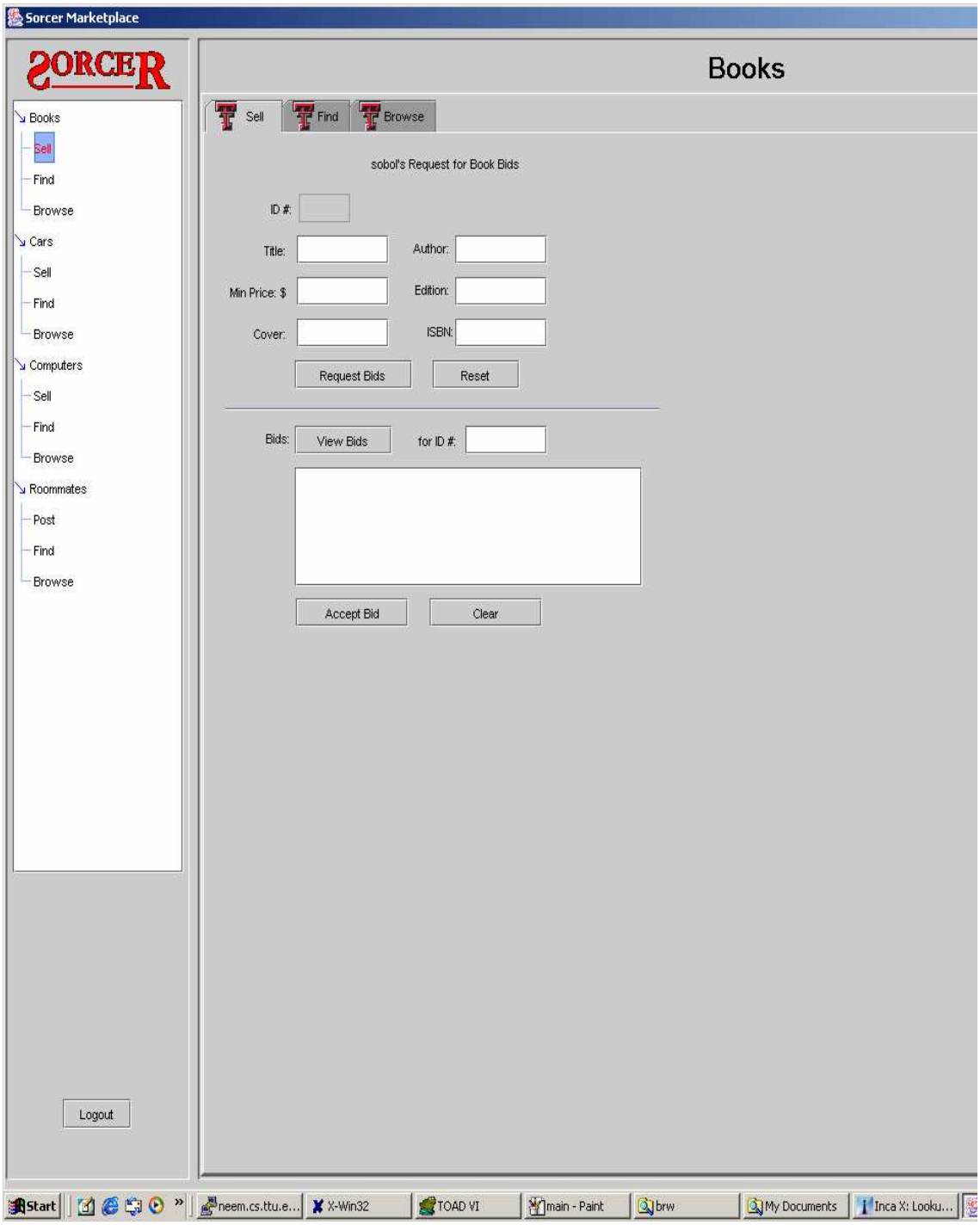


Figure 9.15. Sell Books Interface

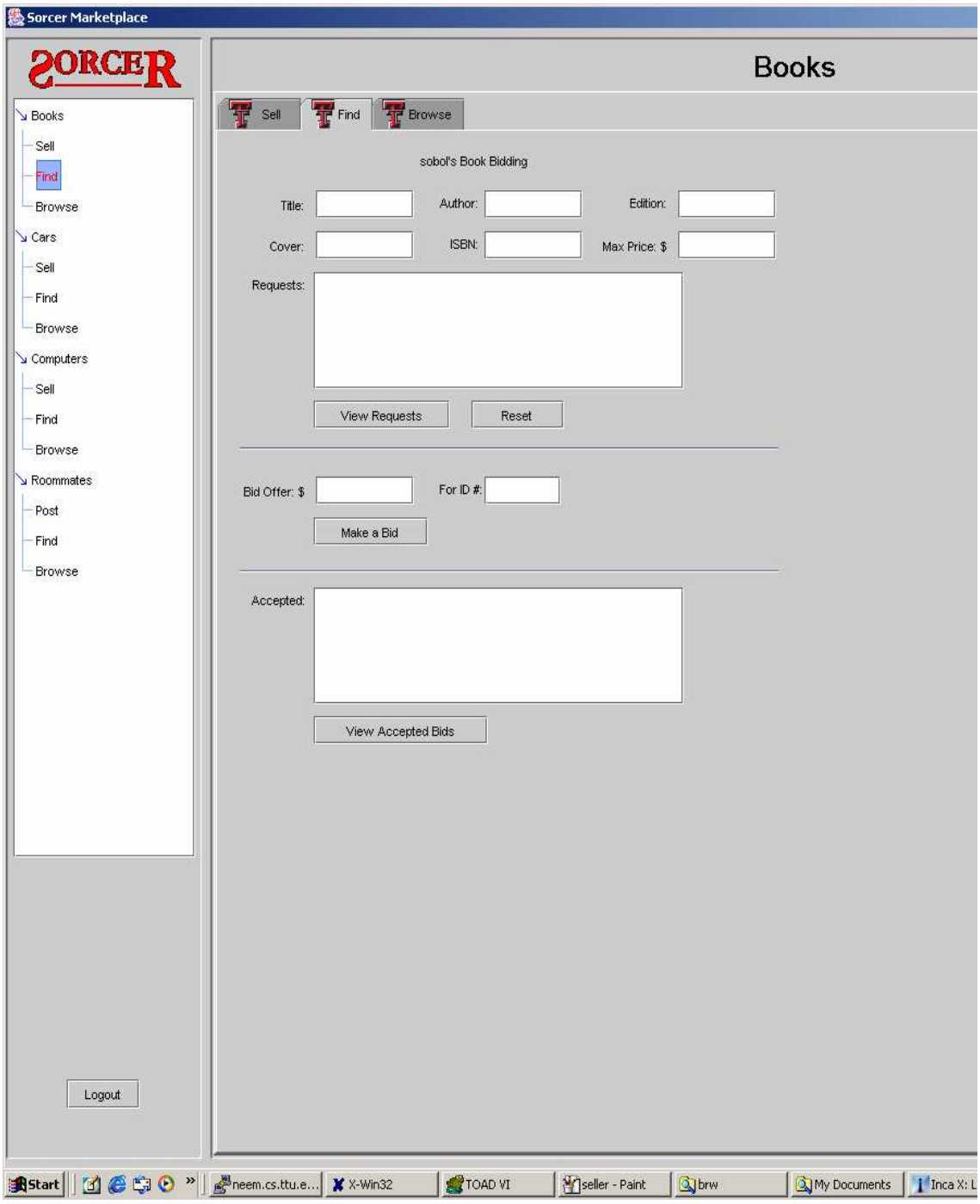


Figure 9.16. Find Books Interface

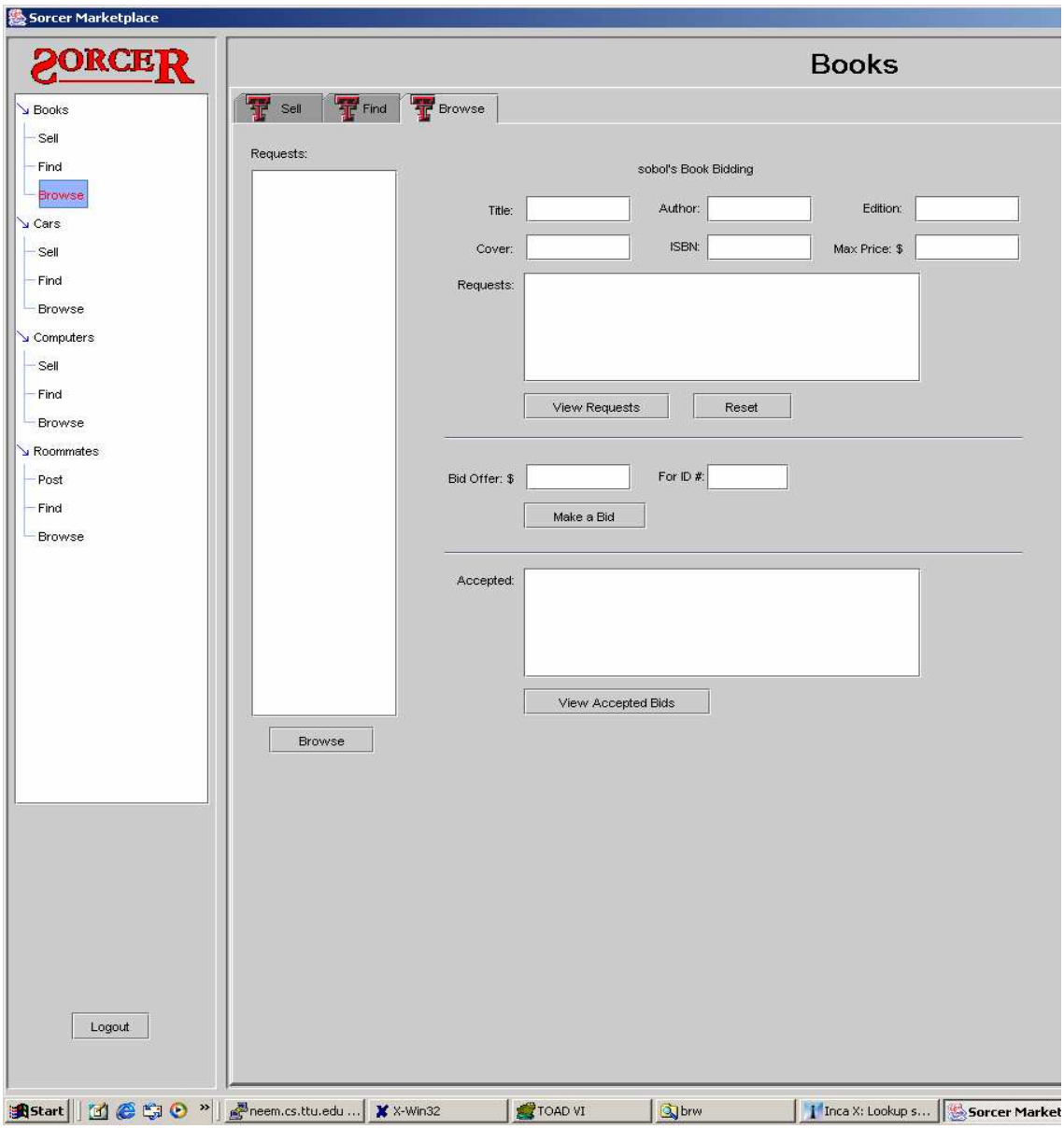


Figure 9.17. Browse Books Interface

Work area for a seller is shown in Figure 9.15. A buyer can perform three operations. He can find the books based on some search criteria. If he finds the book that he wants he can select the book and can make a bid on that book. He can also view if he is winner of any of the books. Work area for buyer is shown in Figure 9.16.

Browse work area for books works in same way as find books work area. It has additional functionality of browsing all the bids that are open in system at that time. A user can then either select a book from a list of all the books or he can narrow down results using find functionality. This is shown in Figure 9.17.

9.9.4 Authentication In Bulletin Board

Authentication is provided using SCAF in Bulletin Board application. To authenticate into the system, a user is required to have a java card plugged into card reader and reader connected to computer. User is then shown with a login frame which requires him to enter the password for the card. If the password is accepted by the card then user is logged into the system and can then acts as a buyer or a seller.

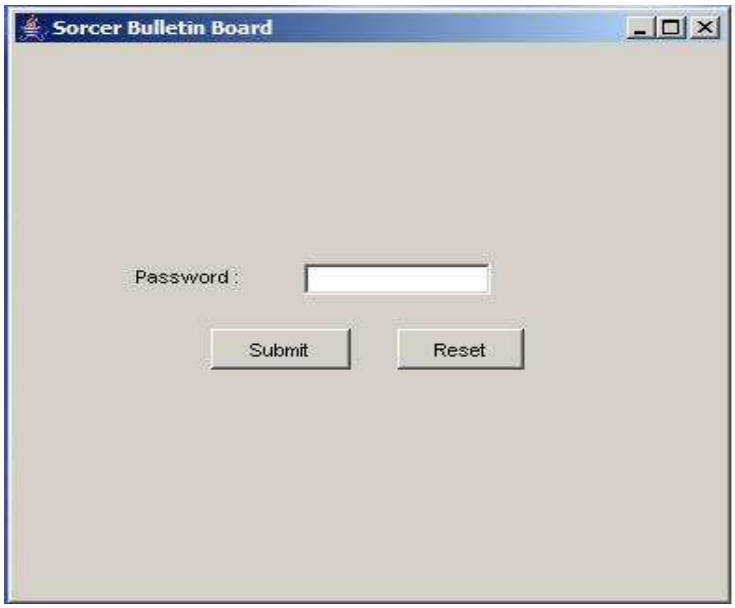


Figure 9.18. Logon Frame

Login frame is shown in Figure 9.18. It has two buttons; submit and reset. A user is given three chances to authenticate. If all three chances fail then application needs to be started again. On the other hand, card can take up to eight unsuccessful attempts. A card is locked if it is presented with wrong password for more than eight times.

9.9.5 Authorization In Bulletin Board

Figure 9.19. Authorization Failure

A user has to have proper credentials issued to him by SORCER certificate authority. If a user has the required credentials on card then he is allowed to perform any action for which access has already been defined for him in policy file. Permissions are granted to the principal name in policy file and hence the principal name from the card has to be sent to framework. The principal name used for authorization is set explicitly in the context for the task that needs to be executed. Authorization can restrict the realm of user, for instance, a user might just be allowed to act as a buyer but he can not act as a seller. When ever a user tries to perform something for which he has not been given any rights then an error message like Figure 9.19 is shown to him. On the hand, if the user has the rights defined for him to perform a function then it is executed and the results are shown to user in the work area of the interface.

9.9.6 Non-Repudiation In Bulletin Board

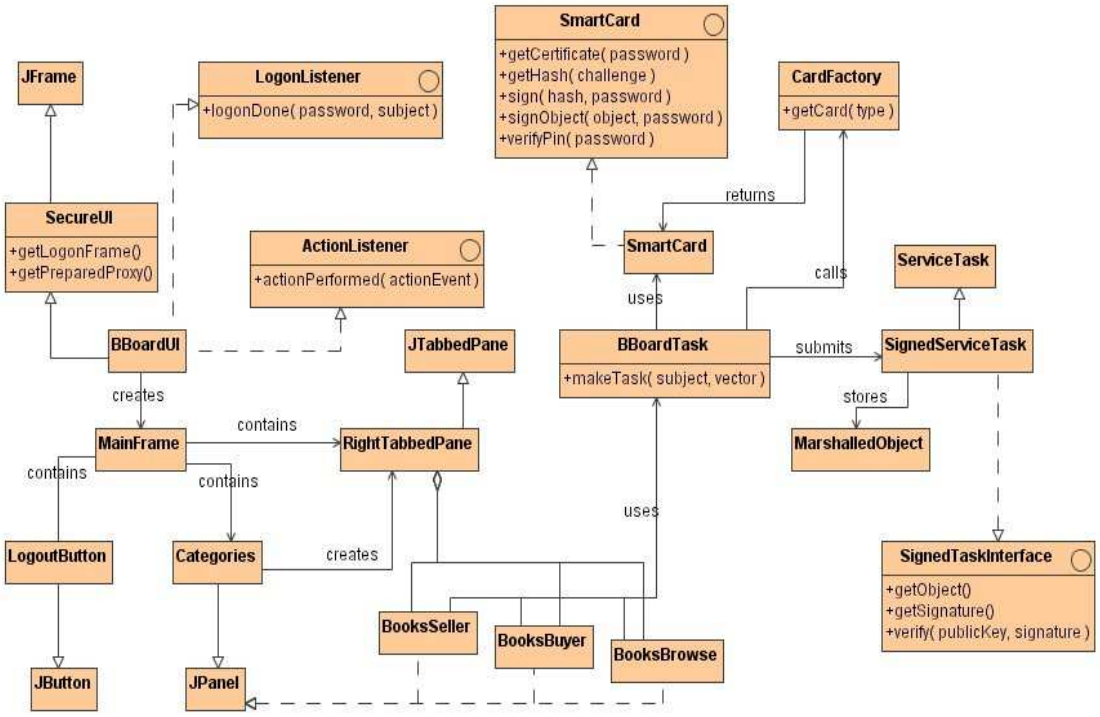


Figure 9.20. Signing Class Diagram

All the functions in SORCER are executed as either a task or a job. A job is a combination of tasks which is divided to tasks before execution. SCAF, as of now, takes only tasks as exertion. For providing non-repudiation functionality all the tasks that are executed are signed first using private key on the card. All the classes that are used for signing task are shown in Figure 9.20.

Signed task is received by SORCER and is then sent to Auditor service. This service stores the task along with the principal name sent with it and date and time in an embedded database. There is an additional application provided with SCAF that be used to verify any repudiation claims.

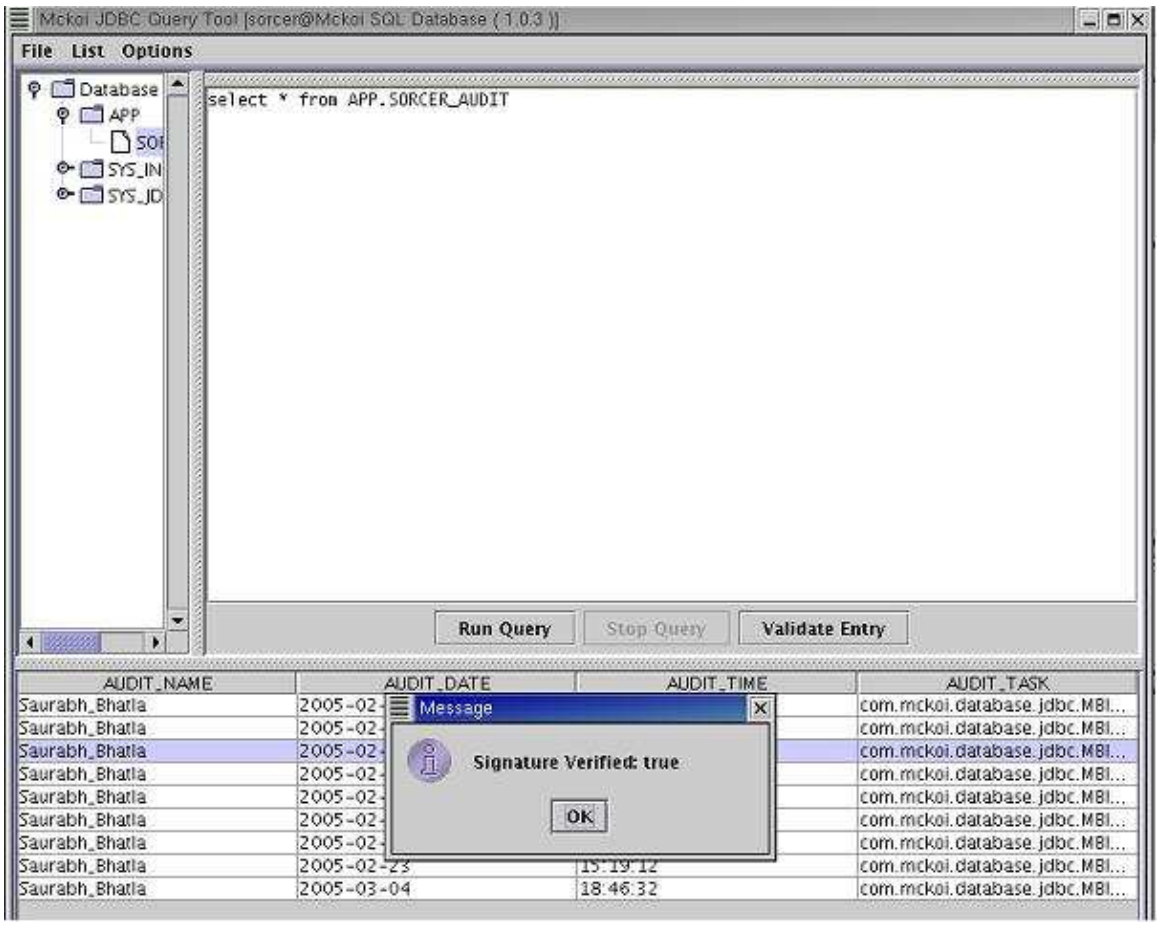


Figure 9.21. Non-Repudiation Interface

Administrator of system can start this application and then select the transaction for which a repudiation claim has been received. This transaction can then be verified using the truststore that contains certificate for all the users of application. Public key from certificates is used to verify if the signed service task when decrypted gives the object that is saved in signed service task. Query tool is shown in Figure 9.21.

9.10 Conclusion

This research has achieved its goal of designing a Smart Card based framework for SORCER that provides user authentication and authorization. This standard security mechanism enforces more consistent security policies, and application developers are freed from the low-level drudgery of building explicit security controls into their software. The security framework allows services anywhere in the world to identify each other, exchange data in encrypted form and to digitally "sign" information in ways that cannot later be repudiated. SCAF is successfully tested to provide integrity and confidentiality for communication between provider and requestor. Smart card ensures the isolation of private key, which enables the protection of sensitive cryptographic data and provides for a secure processing environment. Authentication, authorization and non-repudiation provided by framework are tested successfully using Bulletin Board application.

9.11 Future Research

Providers in SCAF are started using the credentials from the keystore. This framework can be extended to provide Smart Card based start up, where another proxy certificate would be used to make SSL tunnel with the requestor.

It would be interesting to see how Bio Cards can be used for authentication in SCAF. Bio Cards provide authentication based on some biological identity like finger prints. Presently there is no proxy verification in SCAF. Some research can also be done in that realm.

REFERENCES

[1] http://java.sun.com/development/technicalArticles/jini/protocols.html.

[2] Sun homepage. http://www.sun.com/software/jini/faqs/index.xml#q1

[3] Oaks, S. & Wong, H. (2000). Jini in a Nutshell, O'Reilly, ISBN 1-56592-759-1.

[4] http://www.sun.com/software/jini/whitepapers/architecture.html

[5] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java(TM) Development Kit 1.2. In Proceedings of the USENIX Symposium on Internet Technologies and Systems, pages 103-112, Monterey, California, December 1997.

[6] R. Housley, W. Ford, T. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. Request for Comments 2459, Internet Engineering Task Force, January 1999.

[7] T. Ryutov and B.C. Neuman. Access Control Framework for Distributed Applications. Internet Draft, Internet Engineering Task Force, November 1998.

[8] V. Samar and C. Lai. Making Login Services Independent from Authentication Technologies. In Proceedings of the SunSoft

[9] User Authentication And Authorization In The Java(Tm) Platform. Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers

[10] Digital Signatures and PKCS#11 Smart Cards Concepts, Issues and some Programming Details Developer's Conference, March 1996. Rekesh John and Rajesh Parikh.

[11] PKCS #11: Cryptographic Token Interface Standard. RSA Laboratories

[12] PKCS #12: Personal Information Exchange Syntax Standard. RSA Laboratories

[13] PKCS #15: Cryptographic Token Information Format Standard. RSA Laboratories

[14] OpenCA documentation. http://www.openca.org/openca/docs/

[15] OpenCard Framework 1.2, Programmer's Guide

[16] Introduction to JAAS and Java GSS-API Tutorials. http://java.sun.com/j2se/1.4.2/docs/guide/security/jgss/tutorials/

[17] Java Card Technology for Smart Cards. Zhiqun Chen. Addison Wesley

[18] MUSCLE home page. http://www.linuxnet.com

[19] MUSCLE Cryptographic Card Edge Definition for Java1 Enabled Smartcards. David Corcoran and Tommaso Cucinotta.

[20] SSL specification. http://wp.netscape.com/eng/ssl3/3-SPEC.HTM#3

[21] Applied cryptography. http://www.ldv.ei.tum.de/media/files/lehre/itsec/7_PublicKeyInfrastructures.pdf

[22] Mckoi SQL Database. http://mckoi.com/database/UseEmbeddedApp.html

[23] Smart Card Basics. http://www.smartcardbasics.com/

[24] Jini Architecture Specification 2000; Jini.org; Edwards, 2000

[25] Freeman, 1999; Halter, 2002

[26] UML 2.0 specification, Current official version. http://www.uml.org/#UML2.0

APPENDIX

A. SCAF Interfaces

A.1 LogonListener

```
public interface LogonListener
{
        public void logonDone(String password, Subject subject);
}
```

A.2 LoginModule

```
public interface LoginModule
{
        public boolean abort();
        public boolean commit();
        public void initialize (Subject subject, CallbackHandler
        callbackHandler, Map sharedState, Map options);
        public Boolean login();
        public Boolean logout();
}
```

A.3 CallbackHandler

```
public interface CallbackHandler
{
        public void handle (Callback[] callbacks);
}
```

A.4 Callback

```
public interface Callback
{}
```

A.5 SmartCard

```
public interface SmartCard
{
        public boolean verifyPin(String password);
        public byte[] getHash(String challenge);
        public byte[] getCertificate(String password);
        public byte[] sign(byte[] hash,String password);
        public byte[] signObject(Serializable object,String password)
                throws IOException, InvalidKeyException, SignatureException;
}
```

A.6 SignedTaskInterface

```
public interface SignedTaskInterface
{
        public byte[] getSignature();
        public void setSignature(byte[] signature,MarshalledObject mobject)
                throws IOException;
        public Object getObject()
                throws IOException, ClassNotFoundException;
        public boolean verify(PublicKey publickey, Signature signature1)
                throws InvalidKeyException, SignatureException, IOException,
                ClassNotFoundException;
}
```

A.7 Auditor

```
public interface Auditor extends Remote
{
        public void audit(ServiceContext context) throws RemoteException;
}
```

Service-Oriented Computing Environmet: SCAF

SORCER: SCAF	
sorcer.core.provider.auditor	Package containing all the classes for Auditor provider.
sorcer.scaf.auth	Package containing all the classes required by login module.
sorcer.scaf.card	Package containing all the classes required to work with Smart Cards.
sorcer.scaf.ui	Package containing all the classes required to render user interface
sorcer.security.sign	Package containing all the classes for making a signed service task.

sorcer.core.provider.auditor

Interface Auditor

All Superinterfaces:

java.rmi.Remote

All Known Implementing Classes:

AuditorImpl

---

public interface **Auditor**
extends java.rmi.Remote

All implementation of this interface are Service providers in SORCER and are used to save ServiceContext sent to it. In SCAF the ServiceContext contains a SignedServiceTask and a Subject that contains principal name of the user/requestor from the card.

Author:

Saurabh Bhatla

Method Summary

void	**audit**(sorcer.base.ServiceContext context) Audits the information sent in context in a persistent storage.

Method Detail

audit

```
public void audit(sorcer.base.ServiceContext context)
          throws java.rmi.RemoteException
```

Audits the information sent in context in a persistent storage.

Throws:

`java.rmi.RemoteException`

---

sorcer.core.provider.auditor

Class AuditorImpl

```
java.lang.Object
  └sorcer.core.provider.ServiceProvider
      └sorcer.core.provider.SorcerProvider
          └sorcer.core.provider.auditor.AuditorImpl
```

All Implemented Interfaces:

net.jini.admin.Administrable, sorcer.core.AdministratableProvider, Auditor, com.sun.jini.admin.DestroyAdmin, java.util.EventListener, net.jini.admin.JoinAdmin, sorcer.base.MonitorableServicer, sorcer.base.Provider, net.jini.export.ProxyAccessor, java.rmi.Remote, net.jini.core.constraint.RemoteMethodControl, java.io.Serializable, net.jini.security.proxytrust.ServerProxyTrust, net.jini.lookup.ServiceIDListener, sorcer.base.Servicer, sorcer.util.SORCER

---

public class **AuditorImpl**
extends sorcer.core.provider.SorcerProvider
implements Auditor, sorcer.util.SORCER

AuditorImp is implementation of Auditor imterface and is a service provider in SORCER which is used in SCAF to save SignedServiceTasks. It uses Mckoi as an embedded database to save SignedServieTask. It receives SignedServiceTask along with prinicipal name in service context. This principal name is used to identify the rows in the table that stores information about tasks.

A JDBCQueryTool is provided with SCAF that is used to verify any repudiation claims.

See Also:

Serialized Form

Nested Class Summary

Nested classes inherited from class sorcer.core.provider.SorcerProvider
sorcer.core.provider.SorcerProvider.KeepAwake

Field Summary

Fields inherited from class sorcer.core.provider.ServiceProvider
delegate

Fields inherited from interface sorcer.util.SORCER

ADD_DATANODE, ADD_DOMAIN, ADD_JOB_TO_SESSION, ADD_LEAFNODE,
ADD_SUBDOMAIN, ADD_TASK, ADD_TASK_TO_JOB_SAVEAS,
ADD_TASK_TO_JOB_SAVEAS_RUNTIME, APPEND, AS_PROPS, AS_SESSION,
ATTRIBUTE_MODIFIED, BGCOLOR, BROKEN_LINK, CATALOG_CONTENT,
CATALOGER_EVENT, CLEANUP_SESSION, CMPS, Command,
CONTEXT_ATTRIBUTE_VALUES, CONTEXT_ATTRIBUTES, CONTEXT_RESULT, CPS,
CREATION_TIME, DATANODE_FLAG, DELETE_CONTEXT_EVT, DELETE_JOB_EVT,
DELETE_NOTIFICATIONS, DELETE_SESSION, DELETE_TASK, DELETE_TASK_EVT,
DROP_EXERTION, EXCEPTION_IND, EXCEPTIONS, EXERTION_PROVIDER, FALSE,
GET, GET_CONTEXT, GET_CONTEXT_NAMES, GET_FT, GET_JOB,
GET_JOB_NAME_BY_JOB_ID, GET_JOBDOMAIN, GET_JOBNAMES,
GET_NEW_SERVLET_MESSAGES, GET_NOTIFICATIONS_FOR_SESSION,
GET_RUNTIME_JOB, GET_RUNTIME_JOBNAMES, GET_SESSIONS_FOR_USER, GET_TASK,
GET_TASK_NAME_BY_TASK_ID, GET_TASK_NAMES, GETALL_DOMAIN_SUB, IN_FILE,
IN_PATH, IN_SCRIPT, IN_VALUE, IND, IS_NEW, JOB_ID, JOB_NAME, JOB_STATE,
JOB_TASK, MAIL_SEP, MAX_LOOKUP_WAIT, MAX_PRIORITY, META_MODIFIED,
MIN_PRIORITY, MODIFY_LEAFNODE, MSG_CONTENT, MSG_ID, MSG_SOURCE,
MSG_TYPE, NEW_CONTEXT_EVT, NEW_JOB_EVT, NEW_TASK_EVT, NONE,
NORMAL_PRIORITY, NOTIFY_EXCEPTION, NOTIFY_FAILURE, NOTIFY_INFORMATION,
NOTIFY_WARNING, NOTRUNTIME, NULL, OBJECT_DOMAIN, OBJECT_NAME,
OBJECT_OWNER, OBJECT_SCOPE, OBJECT_SUBDOMAIN, Order, OUT_COMMENT,
OUT_FILE, OUT_PATH, OUT_SCRIPT, OUT_VALUE, PERSIST_CONTEXT,
PERSIST_JOB, PERSIST_SORCER_NAME, PERSIST_SORCER_TYPES,
PERSISTENCE_EVENT, POSTPROCESS, PREPROCESS, PRIVATE, PRIVATE_SCOPE,
PROCESS, PROVIDER, PROVIDER_CONTEXT, PUBLIC_SCOPE,
REGISTER_FOR_NOTIFICATIONS, REMOVE_CONTEXT, REMOVE_DATANODE,
REMOVE_JOB, REMOVE_TASK, RENAME_CONTEXT, RENAME_SORCER_NAME,
RESUME_JOB, RUNTIME, SAPPEND, SAVE_TASK_AS, SAVEJOB_AS,
SAVEJOB_AS_RUNTIME, SCRATCH_CONTEXTIDS, SCRATCH_JOBEXERTIONIDS,
SCRATCH_METHODIDS, SCRATCH_TASKEXERTIONIDS, Script, SCRIPT, SELECT,
SELF, SERVICE_EXERTION, SOC_BOOLEAN, SOC_CONTEXT_LINK, SOC_DATANODE,
SOC_DB_OBJECT, SOC_DOUBLE, SOC_FLOAT, SOC_INTEGER, SOC_LONG,
SOC_PRIMITIVE, SOC_SERIALIZABLE, SOC_STRING, SORCER_FOOTER,
SORCER_HEADER, SORCER_HOME, SORCER_INTRO, SORCER_TMP_DIR, SPOSTPROCESS,
SPREPROCESS, SPROCESS, STEP_JOB, STOP_JOB, STOP_TASK,
SUBCONTEXT_CONTROL_CONTEXT_STR, SUSPEND_JOB, SYSTEM_SCOPE, TABLE_NAME,
TASK_COMMAND, TASK_ID, TASK_JOB, TASK_NAME, TASK_PROVIDER, TASK_SCRIPT,
TRUE, UPDATE_CONTEXT, UPDATE_CONTEXT_EVT, UPDATE_DATANODE,
UPDATE_EXERTION, UPDATE_JOB, UPDATE_JOB_EVT, UPDATE_TASK,
UPDATE_TASK_EVT

Constructor Summary

AuditorImpl()
Default Constructor

AuditorImpl(java.lang.String[] args,
com.sun.jini.start.LifeCycle lifeCycle)

Method Summary

void	**audit**(sorcer.base.ServiceContext ctx) Audits the information sent in context in a persistent storage.

Methods inherited from class sorcer.core.provider.SorcerProvider

addLookupAttributes, addLookupGroups, addLookupLocators, destroy, getAdmin, getConstraints, getGrants, getLookupAttributes, getLookupGroups, getLookupLocators, getProxy, getProxyVerifier, getServiceProxy, grant, grantSupported, init, modifyLookupAttributes, removeLookupGroups, removeLookupLocators, setConstraints, setLookupGroups, setLookupLocators, toString

Methods inherited from class sorcer.core.provider.ServiceProvider

doJob, doTask, dropJob, dropTask, fireEvent, getAttributes, getDelegate, getDescription, getGroups, getInfo, getLeastSignificantBits, getMainUIDescriptor, getMethodContexts, getMostSignificantBits, getProperties, getProperty, getProviderID, getProviderName, getScratchDirectory, getScratchURL, hangup, init, init, invokeMethod, invokeMethod, isActive, isValidMethod, isValidTask, loadConfiguration, notifyException, notifyException, notifyExceptionWithStackTrace, notifyFailure, notifyFailure, notifyInformation, notifyWarning, processJob, quit, removeScratchDirectory, restore, resume, service, service0, serviceIDNotify, setProperties, startTiming, step, stop, stopTiming, suspend, update

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface sorcer.base.Provider

fireEvent, getAttributes, getDescription, getGroups, getInfo, getMainUIDescriptor, getMethodContexts, getProperties, getProperty, getProviderID, getProviderName, hangup, init, init, invokeMethod, invokeMethod, isValidMethod, isValidTask, notifyException, notifyException, notifyExceptionWithStackTrace, notifyFailure, notifyFailure, notifyInformation, notifyWarning, restore, setProperties, startTiming, stopTiming, update

Methods inherited from interface sorcer.base.MonitorableServicer

resume, step, stop, suspend

Methods inherited from interface sorcer.base.Servicer
service

Constructor Detail

AuditorImpl

```
public AuditorImpl()
            throws java.rmi.RemoteException
```

Default Constructor

Throws:

`java.rmi.RemoteException` - if remote communication could not be performed

AuditorImpl

```
public AuditorImpl(java.lang.String[] args,
                   com.sun.jini.start.LifeCycle lifeCycle)
            throws java.lang.Exception
```

Method Detail

audit

```
public void audit(sorcer.base.ServiceContext ctx)
```

Audits the information sent in context in a persistent storage.

Specified by:

`audit` in interface `Auditor`

---

sorcer.core.provider.auditor

Class JDBCQueryTool

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └javax.swing.JComponent
              └sorcer.core.provider.auditor.JDBCQueryTool
```

All Implemented Interfaces:

java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable

---

public class **JDBCQueryTool**
extends javax.swing.JComponent

An SQL query tool that allows for queries to be executed to a JDBC driver.

Author:

Tobias Downer, Christophe Nigaud (Treeview - atrap@club-internet.fr)

See Also:

Serialized Form

Nested Class Summary

class	**JDBCQueryTool.DBItem** Inner class for Tables

Nested classes inherited from class javax.swing.JComponent
javax.swing.JComponent.AccessibleJComponent

Nested classes inherited from class java.awt.Container
java.awt.Container.AccessibleAWTContainer

Nested classes inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

Fields inherited from class javax.swing.JComponent

accessibleContext, listenerList, TOOL_TIP_TEXT_KEY, ui, UNDEFINED_CONDITION, WHEN_ANCESTOR_OF_FOCUSED_COMPONENT, WHEN_FOCUSED, WHEN_IN_FOCUSED_WINDOW

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

JDBCQueryTool(com.mckoi.jfccontrols.QueryAgent in_query_agent) Constructs the JComponent.

Method Summary

static void	**main**(java.lang.String[] args) Application start point.

Methods inherited from class javax.swing.JComponent
addAncestorListener, addNotify, addPropertyChangeListener, addPropertyChangeListener, addVetoableChangeListener, computeVisibleRect, contains, createToolTip, disable, enable, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, fireVetoableChange, getAccessibleContext, getActionForKeyStroke, getActionMap, getAlignmentX, getAlignmentY, getAncestorListeners, getAutoscrolls, getBorder, getBounds, getClientProperty, getComponentGraphics, getConditionForKeyStroke, getDebugGraphicsOptions, getDefaultLocale, getGraphics, getHeight, getInputMap, getInputMap, getInputVerifier, getInsets, getInsets, getListeners, getLocation, getMaximumSize, getMinimumSize, getNextFocusableComponent, getPreferredSize, getPropertyChangeListeners, getPropertyChangeListeners, getRegisteredKeyStrokes, getRootPane, getSize, getToolTipLocation, getToolTipText, getToolTipText, getTopLevelAncestor, getTransferHandler, getUIClassID, getVerifyInputWhenFocusTarget, getVetoableChangeListeners, getVisibleRect, getWidth, getX, getY, grabFocus, isDoubleBuffered, isLightweightComponent, isManagingFocus, isMaximumSizeSet, isMinimumSizeSet, isOpaque, isOptimizedDrawingEnabled, isPaintingTile, isPreferredSizeSet, isRequestFocusEnabled, isValidateRoot, paint, paintBorder,

paintChildren, paintComponent, paintImmediately, paintImmediately, paramString, print, printAll, printBorder, printChildren, printComponent, processComponentKeyEvent, processKeyBinding, processKeyEvent, processMouseMotionEvent, putClientProperty, registerKeyboardAction, registerKeyboardAction, removeAncestorListener, removeNotify, removePropertyChangeListener, removePropertyChangeListener, removeVetoableChangeListener, repaint, repaint, requestDefaultFocus, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, resetKeyboardActions, reshape, revalidate, scrollRectToVisible, setActionMap, setAlignmentX, setAlignmentY, setAutoscrolls, setBackground, setBorder, setDebugGraphicsOptions, setDefaultLocale, setDoubleBuffered, setEnabled, setFont, setForeground, setInputMap, setInputVerifier, setMaximumSize, setMinimumSize, setNextFocusableComponent, setOpaque, setPreferredSize, setRequestFocusEnabled, setToolTipText, setTransferHandler, setUI, setVerifyInputWhenFocusTarget, setVisible, unregisterKeyboardAction, update, updateUI

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, addImpl, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getContainerListeners, getFocusTraversalKeys, getFocusTraversalPolicy, getLayout, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusCycleRoot, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, preferredSize, printComponents, processContainerEvent, processEvent, remove, remove, removeAll, removeContainerListener, setFocusCycleRoot, setFocusTraversalKeys, setFocusTraversalPolicy, setLayout, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, createImage, createImage, createVolatileImage, createVolatileImage, disableEvents, dispatchEvent, enable, enableEvents, enableInputMethods, getBackground, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusCycleRootAncestor, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getGraphicsConfiguration, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputContext, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocale, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMouseWheelListeners, getName, getParent, getPeer, getSize, getToolkit, getTreeLock, gotFocus, handleEvent, hasFocus, hide, imageUpdate, inside, isBackgroundSet, isCursorSet,

isDisplayable, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isShowing, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, postEvent, prepareImage, prepareImage, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processMouseEvent, processMouseWheelEvent, remove, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, repaint, repaint, repaint, resize, resize, setBounds, setBounds, setComponentOrientation, setCursor, setDropTarget, setFocusable, setFocusTraversalKeysEnabled, setIgnoreRepaint, setLocale, setLocation, setLocation, setName, setSize, setSize, show, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

JDBCQueryTool

```
public JDBCQueryTool(com.mckoi.jfccontrols.QueryAgent in_query_agent)
```

Constructs the JComponent.

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Application start point.

---

sorcer.scaf.auth

Class CardCallbackHandler

```
java.lang.Object
  └ sorcer.scaf.auth.CardCallbackHandler
```

All Implemented Interfaces:

javax.security.auth.callback.CallbackHandler

---

public class **CardCallbackHandler**
extends java.lang.Object
implements javax.security.auth.callback.CallbackHandler

`CardCallbackHandler` implements CallbackHandler and passes it to underlying security services so that they may interact with the application to retrieve specific authentication data, such as card passwords.

Underlying security services make requests for different types of information by passing individual Callbacks to the `CardCallbackHandler`. The `CardCallbackHandler` implementation decides how to retrieve information depending on the Callbacks passed to it. In SCAF the underlying service needs a password to authenticate a user on card, it uses a `CardPasswordCallback`.

Author:

Saurabh Bhatla

See Also:

`CardPasswordCallback`

Constructor Summary

`CardCallbackHandler``(java.lang.String password)` Constructor that initializes CallbackHandlet with password.

Method Summary

`void`	**`handle`**`(javax.security.auth.callback.Callback[] callbacks)` Invoke an array of Callbacks.
`void`	**`out`**`(java.lang.String str)` Prints debug statements

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

CardCallbackHandler

`public` **`CardCallbackHandler`**`(java.lang.String password)`

Constructor that initializes CallbackHandlet with password.

Parameters:

`password` - user password presented to the system

Method Detail

handle

```
public void handle(javax.security.auth.callback.Callback[] callbacks)
            throws java.io.IOException,

javax.security.auth.callback.UnsupportedCallbackException
```

Invoke an array of Callbacks.

Specified by:

`handle` in interface `javax.security.auth.callback.CallbackHandler`

Parameters:

`callbacks` - an array of Callback objects which contain the information requested by an underlying security service to be retrieved or displayed.

Throws:

`java.io.IOException` - if an input or output error occurs.

`javax.security.auth.callback.UnsupportedCallbackException` - if the implementation of this method does not support one or more of the Callbacks specified in the callbacks parameter.

---

out

`public void` **`out`**`(java.lang.String str)`

Prints debug statements

---

sorcer.scaf.auth

Class CardPasswordCallback

```
java.lang.Object
  └ sorcer.scaf.auth.CardPasswordCallback
```

All Implemented Interfaces:

javax.security.auth.callback.Callback, java.io.Serializable

---

public class **CardPasswordCallback**
extends java.lang.Object
implements javax.security.auth.callback.Callback, java.io.Serializable

CardPasswordCallback allows underlying security services the ability to interact with a calling application to retrieve specific authentication data such as passwords

CardPasswordCallback does not retrieve or display the information requested by underlying security services. CardPasswordCallback simply provide the means to pass such requests to applications, and for applications, if appropriate, to return requested information back to the underlying security services.

Author:

Saurabh Bhatla

See Also:

`CardCallbackHandler`, Serialized Form

Constructor Summary

`CardPasswordCallback``()` Default Constructor

Method Summary

`void`	**`clearPassword`**`()` Clears the password stored in Callback.
`java.lang.String`	**`getPassword`**`()` Returns user password
`void`	**`out`**`(java.lang.String str)` Prints debug statements
`void`	**`setPassword`**`(java.lang.String pass)` Sets user password in Callback.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

CardPasswordCallback

```
public CardPasswordCallback()
```

Default Constructor

Method Detail

getPassword

```
public java.lang.String getPassword()
```

Returns user password

Returns:

user password

---

setPassword

```
public void setPassword(java.lang.String pass)
```

Sets user password in Callback.

---

clearPassword

```
public void clearPassword()
```

Clears the password stored in Callback.

---

out

```
public void out(java.lang.String str)
```

Prints debug statements

---

sorcer.scaf.auth

Class CardLoginModule

```
java.lang.Object
  └ sorcer.scaf.auth.CardLoginModule
```

All Implemented Interfaces:

javax.security.auth.spi.LoginModule

---

public class **CardLoginModule**
extends java.lang.Object
implements javax.security.auth.spi.LoginModule

CardLoginModule authenticates users with a password. This LoginModule checks the user password using card supplied If user successfully authenticates itself, a principal with the user's user name is added to the Subject. This LoginModule has the debug option. If set to true in the login Configuration, debug messages will be output to the output stream, System.out.

Author:

Saurabh Bhatla

See Also:

CardCallbackHandler, CardPasswordCallback

Constructor Summary

CardLoginModule()	

Method Summary

boolean	**abort**() This method is called if the LoginContext's overall authentication failed.
boolean	**commit**() This method is called if the LoginContext's overall authentication succeeded (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules succeeded).
X509Certificate	**getCert**() Retrieves the user's certificate from smart card.
void	**initialize**(javax.security.auth.Subject subject, javax.security.auth.callback.CallbackHandler callbackHandler, java.util.Map sharedState, java.util.Map options) Initialize this LoginModule.
void	**internalCommit**()

	Used by commit to sign proxy certificate and retrieve credentials from smart card.
boolean	**login**() Authenticate the user by getting the password from CardCallbackHandler.
boolean	**logout**() Logout the user.
void	**out**(java.lang.String str) Prints debug statements

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

CardLoginModule

```
public CardLoginModule()
```

Method Detail

initialize

```
public void initialize(javax.security.auth.Subject subject,

javax.security.auth.callback.CallbackHandler callbackHandler,
                       java.util.Map sharedState,
                       java.util.Map options)
```

Initialize this LoginModule.

Specified by:

initialize in interface javax.security.auth.spi.LoginModule

Parameters:

subject - the Subject to be authenticated.

callbackHandler - a CallbackHandler for communicating with the end user (prompting for user names and passwords, for example).

sharedState - shared LoginModule state.

options - options specified in the login Configuration for this particular LoginModule.

---

login

```
public boolean login()
              throws javax.security.auth.login.LoginException
```

Authenticate the user by getting the password from `CardCallbackHandler`.

Specified by:

`login` in interface `javax.security.auth.spi.LoginModule`

Returns:

true in all cases since this LoginModule should not be ignored.

Throws:

`javax.security.auth.login.FailedLoginException` - if the authentication fails.

`javax.security.auth.login.LoginException` - if this LoginModule is unable to perform the authentication.

---

commit

```
public boolean commit()
               throws javax.security.auth.login.LoginException
```

This method is called if the LoginContext's overall authentication succeeded (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules succeeded). If this LoginModule's own authentication attempt succeeded (checked by retrieving the private state saved by the login method), then this method associates a principal, private credentials and public credentials with the Subject located in the LoginModule. If this LoginModule's own authentication attempted failed, then this method removes any state that was originally saved.

Specified by:

`commit` in interface `javax.security.auth.spi.LoginModule`

Returns:

true if this LoginModule's own login and commit attempts succeeded, or false otherwise.

Throws:

`javax.security.auth.login.LoginException` - if the commit fails.

---

abort

```
public boolean abort()
              throws javax.security.auth.login.LoginException
```

This method is called if the LoginContext's overall authentication failed. (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules did not succeed). If this LoginModule's own authentication attempt succeeded (checked by retrieving the private state saved by the login and commit methods), then this method cleans up any state that was originally saved.

Specified by:

`abort` in interface `javax.security.auth.spi.LoginModule`

Returns:

false if this LoginModule's own login and/or commit attempts failed, and true otherwise.

Throws:

`javax.security.auth.login.LoginException` - if the abort fails.

---

logout

```
public boolean logout()
               throws javax.security.auth.login.LoginException
```

Logout the user. This method removes the principal that was added by the commit method.

Specified by:

`logout` in interface `javax.security.auth.spi.LoginModule`

Returns:

true in all cases since this LoginModule should not be ignored.

Throws:

`javax.security.auth.login.LoginException` - if the logout fails.

---

getCert

```
public java.security.cert.X509Certificate getCert()
```

Retrieves the user's certificate from smart card.

Returns:

user certificate

---

internalCommit

```
public void internalCommit()
```

Used by commit to sign proxy certificate and retrieve credentials from smart card.

---

out

```
public void out(java.lang.String str)
```

Prints debug statements

.

sorcer.scaf.auth

Class CardPasswordCallback

```
java.lang.Object
  └ sorcer.scaf.auth.CardPasswordCallback
```

All Implemented Interfaces:

javax.security.auth.callback.Callback, java.io.Serializable

---

public class **CardPasswordCallback**
extends java.lang.Object
implements javax.security.auth.callback.Callback, java.io.Serializable

CardPasswordCallback allows underlying security services the ability to interact with a calling application to retrieve specific authentication data such as passwords

CardPasswordCallback does not retrieve or display the information requested by underlying security services. CardPasswordCallback simply provide the means to pass such requests to applications, and for applications, if appropriate, to return requested information back to the underlying security services.

Author:

Saurabh Bhatla

See Also:

`CardCallbackHandler`, Serialized Form

Constructor Summary

`CardPasswordCallback``()` Default Constructor

Method Summary

`void`	**`clearPassword`**`()` Clears the password stored in Callback.
`java.lang.String`	**`getPassword`**`()` Returns user password
`void`	**`out`**`(java.lang.String str)` Prints debug statements
`void`	**`setPassword`**`(java.lang.String pass)` Sets user password in Callback.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

CardPasswordCallback

```
public CardPasswordCallback()
```

Default Constructor

Method Detail

getPassword

```
public java.lang.String getPassword()
```

Returns user password

Returns:

user password

---

setPassword

```
public void setPassword(java.lang.String pass)
```

Sets user password in Callback.

---

clearPassword

```
public void clearPassword()
```

Clears the password stored in Callback.

---

out

```
public void out(java.lang.String str)
```

Prints debug statements

---

sorcer.scaf.card

Interface SmartCard

All Known Implementing Classes:

JavaCard

public interface **SmartCard**

A generic interface that needs to be implemented by any class that aims to provide smart card functionality.

As of now only one implementation(Java Card) is provided with SCAF. New implementation can be added by implementing this interface.

Method Summary

byte[]	**getCertificate**(java.lang.String password) Returns user certificate from SmartCard
byte[]	**getHash**(java.lang.String challenge) Returns hash of the challenge string presented
byte[]	**sign**(byte[] hash, java.lang.String password) Encrypts the hash presented
byte[]	**signObject**(java.io.Serializable object, java.lang.String password) Encrypts the object presented by first calculating the has for it.
boolean	**verifyPin**(java.lang.String password) Verifies user pin presented.

Method Detail

verifyPin

```
public boolean verifyPin(java.lang.String password)
```

Verifies user pin presented.

Returns:

true if the user is verified and false otherwise

getHash

```
public byte[] getHash(java.lang.String challenge)
```

Returns hash of the challenge string presented

Parameters:

`challenge` - string

Returns:
hash of challenge created using private key from card or null if hash could not be calculated or exception is thrown

---

getCertificate

```
public byte[] getCertificate(java.lang.String password)
```

Returns user certificate from SmartCard

Parameters:

`password` - of user that owns the card

Returns:

certificate read from the card

---

sign

```
public byte[] sign(byte[] hash,
                   java.lang.String password)
```

Encrypts the hash presented

Parameters:

`hash` - of the challenge that needs to be encrypted

`password` - of user that owns the card

Returns:

encrypted hash or null if exception is thrown

---

signObject

```
public byte[] signObject(java.io.Serializable object,
                         java.lang.String password)
                  throws java.io.IOException,
                         java.security.InvalidKeyException,
                         java.security.SignatureException
```

Encrypts the object presented by first calculating the has for it.

Parameters:

`password` - of user that owns the card

Returns:

encrypted object or null if exception is thrown

Throws:

`java.io.IOException` - if the object can not be accessed

`java.security.InvalidKeyException` - if the key read from card is not valid

`java.security.SignatureException` - if the signature is not valid

---

sorcer.scaf.card

Class CardFactory

```
java.lang.Object
  └ sorcer.scaf.card.CardFactory
```

public class **CardFactory**
extends java.lang.Object
Factory to provide various kinds of Smart Card objects

Right now only one kind of Smart Card, Java Card is supported. JavaCard instance can be created by passing the required value. This class provides the static method to get the required instance of Smart Card. All the instances implement `SmartCard` interface and provide a defined set of functions.

Author:
Saurabh Bhatla

See Also:
JavaCard, SmartCard

Field Summary

static int	**JAVA_CARD** JavaCard Type

Constructor Summary

CardFactory()	

Method Summary

static SmartCard	**getCard**(int which) Returns required SmartCard implementation

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

JAVA_CARD

```
public static final int JAVA_CARD
```

JavaCard Type

See Also:

Constant Field Values

Constructor Detail

CardFactory

```
public CardFactory()
```

Method Detail

getCard

```
public static SmartCard getCard(int which)
```

Returns required SmartCard implementation

Parameters:

`which` - type of card instance that is required

Returns:

SmartCard implement if exists othewise null

---

sorcer.scaf.card

Class JavaCard

```
java.lang.Object
  └ sorcer.scaf.card.JavaCard
```

All Implemented Interfaces:

opencard.core.event.CTListener, java.util.EventListener, SmartCard

---

public class **JavaCard**
extends java.lang.Object
implements opencard.core.event.CTListener, SmartCard

JavaCard Implementation of SmartCard. Needs user password to to log into javacard. A window is shown if the card is not plugged in.

Uses PassThruCardService to access CardEdgeApplet residing on the card. Java Card has to have CardEdgeApplet installed as all the commands presented to the card are specific to that applet. For more information on CardEdgeApplet visit www.muscle.org

Provides implementation of all SmartCard function like for password verification, hash generation, encryption (strings and objects) and retrieving user certificates from card.

Author:

Saurabh Bhatla

See Also:

SmartCard

Field Summary

static byte	**CHLG_LEN** lenght of host challenge

Constructor Summary

JavaCard() Default Constructor

Method Summary

void	**cardInserted**(opencard.core.event.CardTerminalEvent ctEvent) Signalled when cars is inserted into the reader
void	**cardRemoved**(opencard.core.event.CardTerminalEvent ctEvent) Signalled when card is removed from the reader

boolean	**getCardIn**() Retuns true if card is in the reader
byte[]	**getCertificate**(java.lang.String password) Returns user certificate from SmartCard
byte[]	**getHash**(java.lang.String challenge) Returns hash of the challenge string presented
protected byte[]	**getResponse**(byte len) Retrieves response apdus from card.
opencard.core.service.SmartCard	**getSmartCard**() Gets Open Card Framework's Smart Card object
void	**getWindow**() Displays the information window to plug in Java Card
void	**out**(java.lang.String str) Prints debug statements
protected byte	**readCertificate**(byte off1, byte off2, byte len) Reads user certificate data from SmartCard
void	**selectApplet**() Selects CardEdgeApplet
void	**shutdown**() Shuts down java card
protected byte	**sigFinal**(byte[] hash) Sends SigFinal command to CardEdgeApplet
protected void	**sigInit**() Sends SigInt command to CardEdgeApplet.
byte[]	**sign**(byte[] hash, java.lang.String password) Encrypts the hash presented
byte[]	**signObject**(java.io.Serializable object, java.lang.String password) Encrypts the object presented by first calculating the has for it.
boolean	**verify**(java.lang.String passwd) Verifies user pin presented.
boolean	**verifyPin**(java.lang.String passwd) USed by verify() to verify user pin presented.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

CHLG_LEN

```
public static byte CHLG_LEN
```

lenght of host challenge

Constructor Detail

JavaCard

```
public JavaCard()
            throws java.lang.Exception
```

Default Constructor

Throws:

`java.lang.Exception` - if initialization could not be done

Method Detail

getWindow

```
public void getWindow()
```

Displays the information window to plug in Java Card

---

cardInserted

```
public void cardInserted(opencard.core.event.CardTerminalEvent ctEvent)
```

Signalled when cars is inserted into the reader

Specified by:

`cardInserted` in interface `opencard.core.event.CTListener`

---

cardRemoved

```
public void cardRemoved(opencard.core.event.CardTerminalEvent ctEvent)
```

Signalled when card is removed from the reader

Specified by:

`cardRemoved` in interface `opencard.core.event.CTListener`

---

getCardIn

```
public boolean getCardIn()
```

Retuns true if card is in the reader

Returns:

true is card is in the reader

---

getSmartCard

```
public opencard.core.service.SmartCard getSmartCard()
```

Gets Open Card Framework's Smart Card object

---

shutdown

```
public void shutdown()
```

Shuts down java card

---

getHash

```
public byte[] getHash(java.lang.String challenge)
```

Returns hash of the challenge string presented

Specified by:

`getHash` in interface `SmartCard`

Parameters:

`challenge` - string

Returns:

hash of challenge created using private key from card or null if hash could not be calculated or exception is thrown

---

verifyPin

```
public boolean verifyPin(java.lang.String passwd)
```

USed by verify() to verify user pin presented.

Specified by:

`verifyPin` in interface `SmartCard`

Returns:

true if the user is verified and false otherwise

---

selectApplet

```
public void selectApplet()
```

Selects CardEdgeApplet

---

verify

```
public boolean verify(java.lang.String passwd)
```

Verifies user pin presented.

Returns:

true if the user is verified and false otherwise

---

sign

```
public byte[] sign(byte[] hash,
                   java.lang.String password)
```

Encrypts the hash presented

Specified by:

sign in interface SmartCard

Parameters:

hash - of the challenge that needs to be encrypted

password - of user that owns the card

Returns:

encrypted hash or null if exception is thrown

---

sigInit

```
protected void sigInit()
```

Sends SigInt command to CardEdgeApplet. Sends parameters like which key, algorith etc to use.

---

sigFinal

```
protected byte sigFinal(byte[] hash)
```

Sends SigFinal command to CardEdgeApplet

Parameters:

hash - of the data that needs to be signed

Returns:

response from CardEdgeApplet

---

getResponse

```
protected byte[] getResponse(byte len)
```

Retrieves response apdus from card. Used by all the methods to get the result of the operation executed

Returns:

data that is read as a response to some previous command apdu sent

---

readCertificate

```
protected byte readCertificate(byte off1,
                               byte off2,
                               byte len)
```

Reads user certificate data from SmartCard

Returns:

certificate data read from the card

---

getCertificate

```
public byte[] getCertificate(java.lang.String password)
```

Returns user certificate from SmartCard

Specified by:

`getCertificate` in interface `SmartCard`

Parameters:

`password` - of user that owns the card

Returns:

certificate read from the card

---

signObject

```
public byte[] signObject(java.io.Serializable object,
                         java.lang.String password)
                  throws java.io.IOException,
                         java.security.InvalidKeyException,
                         java.security.SignatureException
```

Encrypts the object presented by first calculating the has for it.

Specified by:

`signObject` in interface `SmartCard`

Parameters:

`password` - of user that owns the card

Returns:

encrypted object

Throws:

`java.io.IOException` - if the object can not be accessed

`java.security.InvalidKeyException` - if the key read from card is not valid

`java.security.SignatureException` - if the signature is not valid

---

out

```
public void out(java.lang.String str)
```

Prints debug statements

---

sorcer.scaf.ui

Interface LogonListener

public interface **LogonListener**

Listener that is required to be implemented by application classes that need to use SCAF. It is used as channel to pass information from the SCAF framework classes to application that is using SCAF.

Card password would be required by application classes to sign ServiceTask and generate SignedServiceTask

Subject would be used by application classes to be sent in the ServiceContext. This is subject holds the name of the user retrieved from the card.

Author:

Saurabh Bhatla

Method Summary

void	**logonDone**(java.lang.String password, javax.security.auth.Subject subject) Signalled when logon is done successfully.

Method Detail

logonDone

```
public void logonDone(java.lang.String password,
                      javax.security.auth.Subject subject)
```

Signalled when logon is done successfully.

Parameters:

`password` - of the user that was taken by SCAF and is required to be sent to the application

`subject` - that contains user name read from card

sorcer.scaf.ui

Class SecureUI

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Window
              └java.awt.Frame
                  └javax.swing.JFrame
                      └sorcer.scaf.ui.SecureUI
```

All Implemented Interfaces:

javax.accessibility.Accessible, java.awt.image.ImageObserver, java.awt.MenuContainer, javax.swing.RootPaneContainer, java.io.Serializable, javax.swing.WindowConstants

---

public class **SecureUI**
extends javax.swing.JFrame

This is the facade class that is exposed by SCAF to be extended by any application that needs to use SCAF to provide authentication and authorization in SORCER. It gets the proxy object downloaded from the provider and loads the configuration files to be used to make communication channel with the provider. It then checks all the constraints that are specified in the configuration file and throws an exception if any of the constraint is not met.

It uses prepare-minimal.config configuration file which requires Integerity and Confidentiality constraints to be statisfied. It uses SSL channel to provide integrity and confidentiality.

An inner class is used to present the user with the a frame that takes the user password for the card. A subject is created using proxy certificates key pair, which is signed by private key from the card. A user is gives three chances to authenticate himself after which SCAF terminates.

Author:

Saurabh Bhatla

See Also:

CardLoginModule, LogonFrame, Serialized Form

Nested Class Summary

Nested classes inherited from class javax.swing.JFrame
javax.swing.JFrame.AccessibleJFrame

Nested classes inherited from class java.awt.Frame
java.awt.Frame.AccessibleAWTFrame

Nested classes inherited from class java.awt.Window
java.awt.Window.AccessibleAWTWindow

Nested classes inherited from class java.awt.Container
java.awt.Container.AccessibleAWTContainer

Nested classes inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

protected net.jini.config.Configuration	**config** Configuration used to set constraints
protected javax.security.auth.Subject	**loggedSubject** Currently logged in subject
protected java.lang.Object	**preparedProxy** Proxy received after verification

Fields inherited from class javax.swing.JFrame
accessibleContext, EXIT_ON_CLOSE, rootPane, rootPaneCheckingEnabled

Fields inherited from class java.awt.Frame
CROSSHAIR_CURSOR, DEFAULT_CURSOR, E_RESIZE_CURSOR, HAND_CURSOR, ICONIFIED, MAXIMIZED_BOTH, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MOVE_CURSOR, N_RESIZE_CURSOR, NE_RESIZE_CURSOR, NORMAL, NW_RESIZE_CURSOR, S_RESIZE_CURSOR, SE_RESIZE_CURSOR, SW_RESIZE_CURSOR, TEXT_CURSOR, W_RESIZE_CURSOR, WAIT_CURSOR

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface javax.swing.WindowConstants
DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE

Fields inherited from interface java.awt.image.ImageObserver

ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

SecureUI(java.lang.Object obj) Default Constructor.

Method Summary

protected void	**getLogonFrame**(LogonListener listener) Returns LogonFrame that will be used to take password for card
protected java.lang.Object	**getPreparedProxy**() Returns PreparedProxy
javax.security.auth.Subject	**getSubject**() Returns subject that was created after successfull authentication
void	**init**() Initializes SCAF
void	**out**(java.lang.String str) Prints debug statements
protected void	**prepareProxy**(java.lang.Object obj) Prepares the proxy object by performing all validation that would be used to communicate with the provider
protected void	**setPermission**() Sets permission for the prinicipal that was logged in

Methods inherited from class javax.swing.JFrame

addImpl, createRootPane, frameInit, getAccessibleContext, getContentPane, getDefaultCloseOperation, getGlassPane, getJMenuBar, getLayeredPane, getRootPane, isDefaultLookAndFeelDecorated, isRootPaneCheckingEnabled, paramString, processWindowEvent, remove, setContentPane, setDefaultCloseOperation, setDefaultLookAndFeelDecorated, setGlassPane, setJMenuBar, setLayeredPane, setLayout, setRootPane, setRootPaneCheckingEnabled, update

Methods inherited from class java.awt.Frame

addNotify, finalize, getCursorType, getExtendedState, getFrames,

getIconImage, getMaximizedBounds, getMenuBar, getState, getTitle, isResizable, isUndecorated, remove, removeNotify, setCursor, setExtendedState, setIconImage, setMaximizedBounds, setMenuBar, setResizable, setState, setTitle, setUndecorated

Methods inherited from class java.awt.Window

addPropertyChangeListener, addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener, applyResourceBundle, applyResourceBundle, createBufferStrategy, createBufferStrategy, dispose, getBufferStrategy, getFocusableWindowState, getFocusCycleRootAncestor, getFocusOwner, getFocusTraversalKeys, getGraphicsConfiguration, getInputContext, getListeners, getLocale, getMostRecentFocusOwner, getOwnedWindows, getOwner, getToolkit, getWarningString, getWindowFocusListeners, getWindowListeners, getWindowStateListeners, hide, isActive, isFocusableWindow, isFocusCycleRoot, isFocused, isShowing, pack, postEvent, processEvent, processWindowFocusEvent, processWindowStateEvent, removeWindowFocusListener, removeWindowListener, removeWindowStateListener, setCursor, setFocusableWindowState, setFocusCycleRoot, setLocationRelativeTo, show, toBack, toFront

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getContainerListeners, getFocusTraversalPolicy, getInsets, getLayout, getMaximumSize, getMinimumSize, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paint, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, removeAll, removeContainerListener, setFocusTraversalKeys, setFocusTraversalPolicy, setFont, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, createVolatileImage, createVolatileImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBounds, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor,

getDropTarget, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getGraphics, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocation, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isOpaque, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processKeyEvent, processMouseEvent, processMouseMotionEvent, processMouseWheelEvent, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, repaint, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, reshape, resize, resize, setBackground, setBounds, setBounds, setComponentOrientation, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale, setLocation, setLocation, setName, setSize, setSize, setVisible, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object
clone, equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.awt.MenuContainer
getFont, postEvent

Field Detail

config

```
protected net.jini.config.Configuration config
```

Configuration used to set constraints

---

loggedSubject

```
protected javax.security.auth.Subject loggedSubject
```

Currently logged in subject

---

preparedProxy

```
protected java.lang.Object preparedProxy
```

Proxy received after verification

Constructor Detail

SecureUI

```
public SecureUI(java.lang.Object obj)
```

Default Constructor. It uses prepare-minimal.config configuration file which requires Integerity and Confidentiality constraints to be statisfied.

Method Detail

getSubject

```
public javax.security.auth.Subject getSubject()
```

Returns subject that was created after successfull authentication

Returns:

loggedSubject

---

init

```
public void init()
```

Initializes SCAF

---

prepareProxy

```
protected void prepareProxy(java.lang.Object obj)
```

Prepares the proxy object by performing all validation that would be used to communicate with the provider

---

getLogonFrame

```
protected void getLogonFrame(LogonListener listener)
```

Returns LogonFrame that will be used to take password for card

---

getPreparedProxy

```
protected java.lang.Object getPreparedProxy()
```

Returns PreparedProxy

Returns:

prepared proxy

setPermission

```
protected void setPermission()
```

Sets permission for the prinicipal that was logged in

out

```
public void out(java.lang.String str)
```

Prints debug statements

sorcer.security.sign

Interface SignedTaskInterface

All Known Implementing Classes:

SignedServiceTask

---

public interface **SignedTaskInterface**

Interface for making a signed ServiceTask. SignedServiceTask is sent over the communication channel to the provider where ServiceTask is retrieved from it and the SignedServiceTask object is saved in the database along with the prinicipal name sent in context.

Author:

Saurabh Bhatla

See Also:

SignedTaskInterface

Method Summary

java.lang.Object	**getObject**() Returns saved ServiceTask.
byte[]	**getSignature**() Returns encrypted ServiceTask.
void	**setSignature**(byte[] signature, java.rmi.MarshalledObject mobject) Sets the signature and the object whose signature is being sent
boolean	**verify**(java.security.PublicKey publickey, java.security.Signature signature1) Used to verify the signature.

Method Detail

getSignature

```
public byte[] getSignature()
```

Returns encrypted ServiceTask.

Returns:

array containing signed bytes

---

setSignature

```
public void setSignature(byte[] signature,
                         java.rmi.MarshalledObject mobject)
```

```
                throws java.io.IOException
```

Sets the signature and the object whose signature is being sent

Parameters:

`signature` - of ServiceTask object

Throws:

`java.io.IOException` - if the object could not be accessed

---

getObject

```
public java.lang.Object getObject()
                           throws java.io.IOException,
                                  java.lang.ClassNotFoundException
```

Returns saved ServiceTask.

Returns:

object which is an instance of ServiceTask

Throws:

`java.io.IOException` - if the object could not be accessed

`java.lang.ClassNotFoundException` - if the class for object could not be found

---

verify

```
public boolean verify(java.security.PublicKey publickey,
                      java.security.Signature signature1)
               throws java.security.InvalidKeyException,
                      java.security.SignatureException,
                      java.io.IOException,
                      java.lang.ClassNotFoundException
```

Used to verify the signature. Uses PublicKey supplied to decrypt signature and matches it withe object

Parameters:

`publickey` - of the key pair whose private key was used to encrypt the object

Throws:

`java.io.IOException` - if the object could not be accessed

`java.lang.ClassNotFoundException` - if the class for object could not be found

`SignatureExceptiono` - if the signature is not of right format

`java.security.InvalidKeyException` - if the key supplied is not of right format

`java.security.SignatureException`

---

sorcer.security.sign

Class SignedServiceTask

```
java.lang.Object
  └sorcer.core.ExertionImpl
      └sorcer.core.ServiceTask
          └sorcer.security.sign.SignedServiceTask
```

All Implemented Interfaces:

sorcer.base.Exertion, sorcer.core.ExertionState, jgapp.util.GApp, java.io.Serializable, SignedTaskInterface, sorcer.util.SORCER

---

public final class **SignedServiceTask**
extends sorcer.core.ServiceTask
implements SignedTaskInterface, java.io.Serializable

Any task that needs to be saved in a database is sent by encapsulating it in a SignedServiceTask. SignedServiceTask is the implementation of SignedTaskInterface. At the provider if a SignedServiceTask is received, ServiceTask from it is extracted and is used to perform all the method execution. SignedServiceTask is sent to Auditor service along with the subject that is sent with this SignedServiceTask from Service UI to provider. That subject contains the principal name that is used to save this task in database.

Author:

Saurabh Bhatla

See Also:

`SignedTaskInterface`, Serialized Form

Field Summary

Fields inherited from class sorcer.core.ServiceTask
`context`

Fields inherited from class sorcer.core.ExertionImpl
`accessClass, deletedIDs, description, domainID, exceptionCount, exertionID, goodUntilDate, index, isExportControlled, isRuntime, linkCount, linkedIDs, lsbID, methods, mode, monitorSession, msbID, name, ownerID, parentID, principal, priority, project, providerName, runtimeID, scopeCode, selfMode, serviceType, sessionID, startTime, status, subdomainID, subjectID`

Fields inherited from interface sorcer.util.SORCER
`ADD_DATANODE, ADD_DOMAIN, ADD_JOB_TO_SESSION, ADD_LEAFNODE, ADD_SUBDOMAIN, ADD_TASK, ADD_TASK_TO_JOB_SAVEAS,`

ADD_TASK_TO_JOB_SAVEAS_RUNTIME, APPEND, AS_PROPS, AS_SESSION,
ATTRIBUTE_MODIFIED, BGCOLOR, BROKEN_LINK, CATALOG_CONTENT,
CATALOGER_EVENT, CLEANUP_SESSION, CMPS, Command,
CONTEXT_ATTRIBUTE_VALUES, CONTEXT_ATTRIBUTES, CONTEXT_RESULT, CPS,
CREATION_TIME, DATANODE_FLAG, DELETE_CONTEXT_EVT, DELETE_JOB_EVT,
DELETE_NOTIFICATIONS, DELETE_SESSION, DELETE_TASK, DELETE_TASK_EVT,
DROP_EXERTION, EXCEPTION_IND, EXCEPTIONS, EXERTION_PROVIDER, FALSE,
GET, GET_CONTEXT, GET_CONTEXT_NAMES, GET_FT, GET_JOB,
GET_JOB_NAME_BY_JOB_ID, GET_JOBDOMAIN, GET_JOBNAMES,
GET_NEW_SERVLET_MESSAGES, GET_NOTIFICATIONS_FOR_SESSION,
GET_RUNTIME_JOB, GET_RUNTIME_JOBNAMES, GET_SESSIONS_FOR_USER, GET_TASK,
GET_TASK_NAME_BY_TASK_ID, GET_TASK_NAMES, GETALL_DOMAIN_SUB, IN_FILE,
IN_PATH, IN_SCRIPT, IN_VALUE, IND, IS_NEW, JOB_ID, JOB_NAME, JOB_STATE,
JOB_TASK, MAIL_SEP, MAX_LOOKUP_WAIT, MAX_PRIORITY, META_MODIFIED,
MIN_PRIORITY, MODIFY_LEAFNODE, MSG_CONTENT, MSG_ID, MSG_SOURCE,
MSG_TYPE, NEW_CONTEXT_EVT, NEW_JOB_EVT, NEW_TASK_EVT, NONE,
NORMAL_PRIORITY, NOTIFY_EXCEPTION, NOTIFY_FAILURE, NOTIFY_INFORMATION,
NOTIFY_WARNING, NOTRUNTIME, NULL, OBJECT_DOMAIN, OBJECT_NAME,
OBJECT_OWNER, OBJECT_SCOPE, OBJECT_SUBDOMAIN, Order, OUT_COMMENT,
OUT_FILE, OUT_PATH, OUT_SCRIPT, OUT_VALUE, PERSIST_CONTEXT,
PERSIST_JOB, PERSIST_SORCER_NAME, PERSIST_SORCER_TYPES,
PERSISTENCE_EVENT, POSTPROCESS, PREPROCESS, PRIVATE, PRIVATE_SCOPE,
PROCESS, PROVIDER, PROVIDER_CONTEXT, PUBLIC_SCOPE,
REGISTER_FOR_NOTIFICATIONS, REMOVE_CONTEXT, REMOVE_DATANODE,
REMOVE_JOB, REMOVE_TASK, RENAME_CONTEXT, RENAME_SORCER_NAME,
RESUME_JOB, RUNTIME, SAPPEND, SAVE_TASK_AS, SAVEJOB_AS,
SAVEJOB_AS_RUNTIME, SCRATCH_CONTEXTIDS, SCRATCH_JOBEXERTIONIDS,
SCRATCH_METHODIDS, SCRATCH_TASKEXERTIONIDS, Script, SCRIPT, SELECT,
SELF, SERVICE_EXERTION, SOC_BOOLEAN, SOC_CONTEXT_LINK, SOC_DATANODE,
SOC_DB_OBJECT, SOC_DOUBLE, SOC_FLOAT, SOC_INTEGER, SOC_LONG,
SOC_PRIMITIVE, SOC_SERIALIZABLE, SOC_STRING, SORCER_FOOTER,
SORCER_HEADER, SORCER_HOME, SORCER_INTRO, SORCER_TMP_DIR, SPOSTPROCESS,
SPREPROCESS, SPROCESS, STEP_JOB, STOP_JOB, STOP_TASK,
SUBCONTEXT_CONTROL_CONTEXT_STR, SUSPEND_JOB, SYSTEM_SCOPE, TABLE_NAME,
TASK_COMMAND, TASK_ID, TASK_JOB, TASK_NAME, TASK_PROVIDER, TASK_SCRIPT,
TRUE, UPDATE_CONTEXT, UPDATE_CONTEXT_EVT, UPDATE_DATANODE,
UPDATE_EXERTION, UPDATE_JOB, UPDATE_JOB_EVT, UPDATE_TASK,
UPDATE_TASK_EVT

Fields inherited from interface jgapp.util.GApp

ACL_CMD, ACL_FOROBJECT, ACL_ID, ACL_ISAUTHORIZED, ACL_MODE, ACL_OBJID,
ACL_OBJNAME, ACL_OBJTYPE, ACL_OWNER, ACL_PERMISSIONS, ACL_PID,
ACL_PNAME, ACL_PTYPE, ACL_ROLES, ADD_DOCUMENT, ADD_FOLDER, ADD_GROUP,
ADD_PERMISSION, ADD_ROLE, ADD_USER, ADOC, ADRAFT, AOWNER,
APPROVAL_REJECT_REASON, APPROVAL_STATUS, APPROVE_DOC, APPROVED_DOC,
ASSIGN_TO_DOC, ATTACHED, ATYPE, AUTHORIZE, AUTHORIZE_UPLOAD,
BUFFER_ACL, CACL, CADD, CALL, CAPPROVAL, CDELETE, CDOC, CDOCUMENT,
CDRAFT, CEMAIL, CFOLDER, CGROUP, CHANGEPASSWD, CONFIDENTIAL, COWNER,
CPERMISSION, CREAD, CROLE, CUPDATE, CUSER, CVER, CVERSION, CVIEW, DACL,
DACLASS, DAGROUP, DATA, DCONTEXT, DCONTEXTOID, DCVOID, DDDATE, DDESC,
DECONTROL, DELETE_DOCUMENT, DELETE_DRAFT, DELETE_FOLDER,

DELETE_FOLDERS, DELETE_GROUP, DELETE_PERMISSION, DELETE_REVIEW, DELETE_ROLE, DELETE_USER, DELETE_VERSION, DELETED, delim, DGROUPS, DGUDATE, DISPATCH_CMD, DLUDATE, DMEMBERS, DNAME, DNEGPERMS, DO_JOB, DO_TASK, DOC, DOID, DOWNER, DPOSPERMS, DRGROUP, DROLES, DUSER, DWDATE, EXEC MANDATE, EXECCMD, EXECDEFAULT, EXECPQUERY, EXECPREPQUERY, EXECPREPUPDATE, EXECPROVIDER, EXECQUERY, EXECUPDATE, FACLASS, FDESC, FECONTROL, FNAME, FOID, FOLDER_CLOSE, FOLDER_LEAF, FOLDER_OPEN, FOWNER, FPARENT, FPATH, FREEZE_FOLDER, FU_CLASS_ACCESS, FU_CONTEXT_ID, FU_DESC, FU_EXPORT_CONTROL, FU_FILE, FU_FILE_SIZE, FU_MIME_TYPE, FU_MODIFIER, FU_USER, FUPLOAD, GAPP_CMD_END, GENERATE_HTML, GET_ACL, GET_DIRECTORIES, GET_DOCUMENT, GET_DRAFT, GET_GAPP_PRINCIPAL, GET_GAPPACL, GET_ROLES, GET_SSO_PRINCIPAL, GNAME, GOEMAIL, GOFIRST, GOID, GOLAST, GOLOGIN, GOPHONE, GREEN, GROUP, IS_ALIVE, ISVIEW_FOLDER, LIST_DIRECTORIES, LIST_FILES, LOAD_GROUPS, LOAD_PERMISSION, LOAD_ROLES, LOCK_FOLDER, LOG, LOGIN, MAKE_CURRENT, MBCC, MCC, metaSep, MFROM, MODIFIED, MOVE_DOCUMENT, MOVE_FOLDER, MSIZE, MSUBJECT, MTEXT, MTO, NEW, NONE, OBJECT_CMD_START, OPEN_FOLDER, OWNER, PENDING, PID, POPERATION, PREPROCESS_DOCDESC, PSIGN, PTYPE, PUBLIC, RED, REDIRECT, RESTORE_OBJECT, RID, ROID, ROLE, ROPERATION, ROTYPE, RPERMISSION, RPERMISSION_ID, RPERMISSION_OBJTYPE, RPERMISSION_SIGN, RROLE, SECRET, seed, SEND_MAIL, SENSITIVE, sep, sepChar, SERVICE_CMD_START, SERVLET_UPDATE, SQUARE_GREEN, SQUARE_NONE, STATUS, STORE_OBJECT, SUBFOLDERS, SUBSCRIBE_TO_FOLDER, SUPDATE, U_ACTION, U_ARG1, U_ARG2, U_ARG3, UACLASS, UECONTROL, UEMAIL, UFIRST, ULAST, ULOGIN, UNMODIFIED, UOID, UPASS, UPDATE, UPDATE_DOCUMENT, UPDATE_DRAFT, UPDATE_FOLDER, UPDATE_GROUP, UPDATE_PERMISSION, UPDATE_REVIEW, UPDATE_ROLE, UPDATE_USER, UPDATE_VERSION, UPHONE, UPLOAD_ATTACH, UPLOAD_DESCRIPTOR, UPLOAD_DOC, UPLOAD_DRAFT, UPLOAD_END, UPLOAD_REVIEW, UPLOAD_VERSION, UROLE, USERS, USSO, USSOUID, VALIDATE_APPROVAL, VALIDATE_REVIEW, XACCESS_NAME, XACLASS, XCOMMENTS, XCONTEXT, XCONTEXTOID, XDATE, XECONTROL, XOID, XOWNER, XOWNEROID, XVERSION, YELLOW

Fields inherited from interface sorcer.core.ExertionState

DONE, ERROR, FAILED, INITIAL, INSPACE, INVALID_CMD, LOCK_ERROR, RUNNING, STOPPED, SUSPENDED, TRANSACTION_ERROR

Constructor Summary

SignedServiceTask(java.lang.String name, java.lang.String description, sorcer.core.ServiceMethod[] methods)
Constructor to an instance of SignedServiceTask

Method Summary

java.lang.Object	**getObject**() Returns saved ServiceTask.
byte[]	**getSignature**() Sets the signature and the object whose signature is being

	sent
void	**setSignature**(byte[] signature, java.rmi.MarshalledObject mobject) Returns encrypted ServiceTask.
boolean	**verify**(java.security.PublicKey publickey, java.security.Signature signature1) Used to verify the signature.

Methods inherited from class sorcer.core.ServiceTask
addException, doIt, equals, getContext, getContextName, isJob, isTask, job, sc, setContext, setOwnerID, task, toString

Methods inherited from class sorcer.core.ExertionImpl
addMethod, compareByIndex, contextToString, copyValues, getAccessClass, getDeletedIDs, getDescription, getDomainID, getExceptionCount, getGoodUntilDate, getID, getIndex, getLinkCount, getLinkedIDs, getLsbID, getMethod, getMethods, getMode, getMsbID, getName, getOwnerID, getParentID, getPrincipal, getPriority, getProject, getProviderName, getRuntimeID, getScopeCode, getSelfMode, getServiceType, getSessionID, getStatus, getSubdomainID, getSubjectID, isEntry, isExecutable, isExportControlled, isExportControlled, isLinked, isModified, isOrder, isRuntime, isRuntime, isScript, modified, removeMethod, selfModified, setAccessClass, setDeletedIDs, setDescription, setDomainID, setExceptionCount, setGoodUntilDate, setID, setIndex, setLinkCount, setLinkedIDs, setLsbID, setMethods, setMode, setMsbID, setName, setParentID, setPrincipal, setPriority, setProject, setProviderName, setRuntimeID, setScopeCode, setSelfMode, setServiceType, setSessionID, setStatus, setSubdomainID, setSubjectID

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

SignedServiceTask

```
public SignedServiceTask(java.lang.String name,
                         java.lang.String description,
                         sorcer.core.ServiceMethod[] methods)
```

Constructor to an instance of SignedServiceTask

Parameters:

`name` - task name

`description` - task description

`methods` - array of ServiceMethods that need to be executed

Method Detail

setSignature

```
public void setSignature(byte[] signature,
                         java.rmi.MarshalledObject mobject)
                  throws java.io.IOException
```

Returns encrypted ServiceTask.

Specified by:

`setSignature` in interface `SignedTaskInterface`

Parameters:

`signature` - of ServiceTask object

Returns:

array containing signed bytes

Throws:

`java.io.IOException` - if the object could not be accessed

---

getSignature

```
public byte[] getSignature()
```

Sets the signature and the object whose signature is being sent

Specified by:

`getSignature` in interface `SignedTaskInterface`

Returns:

array containing signed bytes

Throws:

`java.io.IOException` - if the object could not be accessed

---

getObject

```
public java.lang.Object getObject()
                           throws java.io.IOException,
                                  java.lang.ClassNotFoundException
```

Returns saved ServiceTask.

Specified by:

`getObject` in interface `SignedTaskInterface`

Returns:

object which is an instance of ServiceTask

Throws:

`java.io.IOException` - if the object could not be accessed

`java.lang.ClassNotFoundException` - if the class for object could not be found

---

verify

```
public boolean verify(java.security.PublicKey publickey,
                      java.security.Signature signature1)
               throws java.security.InvalidKeyException,
                      java.security.SignatureException,
                      java.io.IOException,
                      java.lang.ClassNotFoundException
```

Used to verify the signature. Uses PublicKey supplied to decrypt signature and matches it withe object

Specified by:

verify in interface SignedTaskInterface

Parameters:

publickey - of the key pair whose private key was used to encrypt the object

Throws:

java.io.IOException - if the object could not be accessed

java.lang.ClassNotFoundException - if the class for object could not be found

SignatureExceptiono - if the signature is not of right format

java.security.InvalidKeyException - if the key supplied is not of right format

java.security.SignatureException

---

sorcer.security.sign

Class TaskAuditor

```
java.lang.Object
  └ sorcer.security.sign.TaskAuditor
```

public class **TaskAuditor**
extends java.lang.Object

All secure data or transactions in SCAF are saved using an Auditor Service. This class starts a new worker thread that looks for Auditor Service and then calls auditor method of that to save the task submitted.

Author:

Saurabh Bhatla

See Also:

SignedTaskInterface, SignedServiceTask, TaskAuditor.AuditThread

Nested Class Summary

protected class	**TaskAuditor.AuditThread** Inner class of TaskAuditor< whih starts a new thread to start looking for Auditor Provider.

Field Summary

protected sorcer.core.Auditor	**auditor** Auditor Service Provider

Constructor Summary

TaskAuditor()

Method Summary

void	**audit**(SignedServiceTask task) Audits the SignedServiceTask.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

auditor

```
protected sorcer.core.Auditor auditor
```

Auditor Service Provider

Constructor Detail

TaskAuditor

```
public TaskAuditor()
```

Method Detail

audit

```
public void audit(SignedServiceTask task)
```

Audits the SignedServiceTask.

Parameters:

`task` - a signed service task that needs to be saved.

C. CardEdge Commands

Command Name	S/R	INS (hex)	P1	P2	P3	DATA
Key Handling Commands						
GenerateKeyPair	S	30	Prv Key N.	Pub Key N	Size	Gen Params
ImportKey	S	32	Key N.	0x00	Size	Imp Params
ExportKey	S	34	Key N.	0x00	Size	Exp Params
ComputeCrypt	S	36	Key N.	Operation	Size	Ext Data
ExtAuthenticate	S	38	Key N.	0x00	Size	Ext Data
ListKeys	R	3A	Seq Option	0x00	0x0B	-
Pin Related Commands						
CreatePIN	S	40	PIN N.	Max Attempts	Size	PIN Params
VerifyPIN	S	42	PIN N.	0x00	Size	PIN Code
ChangePIN	S	44	PIN N.	0x00	Size	Params
UnblockPIN	R	46	PIN N.	0x00	Size	Unblock Code
ListPIN	R	48	0x00	0x00	0x02	-
Object Related Commands						
CreateObject	S	5A	0x00	0x00	0x0E	Create ID
DeleteObject	S	52	0x00	Zero Flag	0x04	ObjectID
WriteObject	S	54	0x00	0x00	Size	Params
ReadObject	S/R	56	0x00	0x00	Size	Params
ListObject	R	58	Seq Option	0x00	0x0E	-
Other						
LogOutAll	S	60	0x00	0x00	0x02	0x0000
GetChallenge	S	62	0x00	Output Data Location	Size	Chall. Parameters
GetStatus	R	3C	0x00	0x00	0x10	-
IsoVerify	S	20	0x00	PIN N.	Size	PIN Code
IsoGetResponse	R	C0	0x00	0x00	Expected Size	-

D. CardEdge Error Codes

Return Codes (Status Words)		
Value	Symbolic Name	Description
90 00	SW_SUCCESS (ISO)	Operation successfully completed
9C 01	SW_NO_MEMORY_LEFT	Insufficient memory onto the card to complete the operation
9C 02	SW_AUTH_FAILED	Unsuccessful authentication. Multiple consecutive failures cause the identity to block
9C 03	SW_OPERATION_NOT_ALLOWED	Operation not allowed because of the internal state of the Applet internal state of the Applet
9C 05	SW_UNSUPPORTED_FEATURE	The requested feature is not supported either by the card or by the Applet
9C 06	SW_UNAUTHORIZED	Logged in identities don't have enough privileges for the requested operation
9C 07	SW_OBJECT_NOT_FOUND	An object either explicitly or implicitly involved in the operation was not found
9C 08	SW_OBJ_EXISTS	Object already exists

9C 09	SW_INCORRECT_ALG	Input data to the command contained an invalid algorithm
9C 0B	SW_SIGNATURE_INVALID	The signature provided in a verify operation was incorrect
9C 0C	SW_IDENTITY_BLOCKED	Authentication operation not allowed because specified identity is blocked
9C 0D	SW_UNSPECIFIED_ERROR	An error occurred. No further information is given.
9C 0E	SW_INVALID_PARAMETER	Input data provided either in the APDU or by means of the input objectis invalid
9C 10	SW_INCORRECT_P1	Incorrect P1 value
9C 11	SW_INCORRECT_P2	Incorrect P2 value
9C 12	SW_INCORRECT_LE	When receiving data from the card, expected length is not correct.
63 00	SW_INVALID_AUTH (ISO)	Unsuccessful authentication (for an ISO Verify). Multiple consecutive failures cause the PIN to block
69 83	SW_AUTH_BLOCKED (ISO)	The PIN referenced into an ISO Verify command is blocked
6A 86	SW_INCORRECT_P1P2 (ISO)	Incorrect values of either P1 or P2 parameter or both of them
6D 00	SW_ERROR_INS (ISO)	Instruction code not recognized